A COMPLETE COURSE

Reinforcement Learning

From First Principles to Research

2026 Edition
Research_RL Knowledge Center

Contents

Preface . . . . . **xiii**
Who this book is for xiii
What makes this book different xiii
How the book was made xiv

How to Use This Book . . . . . **xv**
Three tracks xv
What to read first xv
Suggested routes xvi
Labs, exercises and solutions xvii
Conventions xvii

Notation . . . . . **xix**
The interaction xix
Markov decision processes xix
Policies, returns and values xx
Learning xx
Bandits (Chapter 2) xxi
Language models (Chapter 22) xxi
The Trail xxii
Reading levels xxii

Part I Foundations

1 What Is Reinforcement Learning? **3**
1.1 Learning from consequences 4
1.2 The agent–environment loop 4
1.3 A first example: the Trail 6
1.4 What makes reinforcement learning hard 9
1.5 Reinforcement learning among the learning problems 10
1.6 Inside an agent: policies, values and models 11
1.7 Learning by trial and error: a first experiment 12
1.8 Why the best policy of the Trail is deterministic ●● 13
1.9 What is a reward? ●●● 14
1.10 A short history 15
1.11 How this book is organized 16
1.12 Summary 18
1.13 Further Reading 19
1.14 Exercises 19

2 Bandits: Learning to Choose **23**
2.1 The multi-armed bandit problem 24
2.2 Estimating action values 26

2.3 Greedy and ε-greedy action selection . . . 28
2.4 Upper confidence bounds . . . 31
2.5 Thompson sampling . . . 35
2.6 Gradient bandits . . . 37
2.7 Bandits in practice . . . 40
2.8 Regret analysis of UCB1 ●● . . . 41
2.9 Lower bounds ●● . . . 44
2.10 Contextual bandits ●● . . . 46
2.11 Beyond the basic bandit ●●● . . . 48
2.12 Summary . . . 50
2.13 Further Reading . . . 50
2.14 Exercises . . . 51

3 Markov Decision Processes . . . 55
3.1 From bandits to sequential decisions . . . 56
3.2 Markov decision processes . . . 57
3.3 Policies . . . 60
3.4 Returns and discounting . . . 60
3.5 Value functions . . . 62
3.6 The Bellman expectation equations . . . 63
3.7 Optimality . . . 67
3.8 Existence of optimal policies ●● . . . 72
3.9 Occupancy measures and the performance difference lemma ●● . . . 73
3.10 Other objectives ●● . . . 76
3.11 What MDPs leave out . . . 79
3.12 Summary . . . 80
3.13 Further Reading . . . 80
3.14 Exercises . . . 81

4 Dynamic Programming . . . 85
4.1 Planning with a known model . . . 86
4.2 Policy evaluation . . . 86
4.3 Bellman operators and contraction . . . 88
4.4 Policy improvement . . . 92
4.5 Value iteration . . . 95
4.6 Generalized policy iteration . . . 98
4.7 How good is a greedy policy? ●● . . . 100
4.8 Convergence rates and complexity ●● . . . 103
4.9 Linear programming ●●● . . . 104
4.10 Finite horizons and beyond tables . . . 106
4.11 Summary . . . 108
4.12 Further Reading . . . 108
4.13 Exercises . . . 109

Part II Learning from Experience

5 Monte Carlo Methods . . . 115
5.1 Learning from complete episodes . . . 116
5.2 Incremental implementation and step sizes . . . 120
5.3 Monte Carlo control . . . 121

5.4 Off-policy prediction with importance sampling . . . 126
5.5 The variance of importance sampling ●● . . . 131
5.6 Statistical properties of Monte Carlo estimates ●● . . . 134
5.7 Monte Carlo beyond tables ●●● . . . 137
5.8 Summary . . . 138
5.9 Further Reading . . . 139
5.10 Exercises . . . 139

6 Temporal-Difference Learning . . . 143
6.1 TD prediction . . . 144
6.2 Why TD works and when it beats Monte Carlo . . . 148
6.3 SARSA: on-policy TD control . . . 151
6.4 Q-learning: off-policy TD control . . . 152
6.5 Expected SARSA and the family of one-step methods . . . 153
6.6 Cliff walking . . . 155
6.7 Maximization bias and Double Q-learning . . . 157
6.8 Terminal states, time limits and bootstrapping . . . 158
6.9 Convergence of TD and Q-learning ●● . . . 160
6.10 Finite-sample behavior ●●● . . . 162
6.11 TD in the brain ●● . . . 163
6.12 Summary . . . 164
6.13 Further Reading . . . 165
6.14 Exercises . . . 165

7 Multi-Step Methods and Eligibility Traces . . . 171
7.1 Between Monte Carlo and TD: n-step returns . . . 172
7.2 n-step SARSA . . . 177
7.3 Off-policy n-step learning . . . 178
7.4 The λ-return . . . 181
7.5 Eligibility traces and TD(λ) . . . 183
7.6 Forward–backward equivalence ●● . . . 186
7.7 Control with eligibility traces ●● . . . 190
7.8 Off-policy traces with guarantees: Retrace and V-trace ●●● . . . 192
7.9 Multi-step targets in deep reinforcement learning . . . 195
7.10 Summary . . . 196
7.11 Further Reading . . . 197
7.12 Exercises . . . 198

8 Planning and Learning with Models . . . 203
8.1 Models and planning . . . 204
8.2 Dyna: integrating planning, acting and learning . . . 206
8.3 Prioritized sweeping . . . 210
8.4 Expected versus sample updates . . . 212
8.5 Decision-time planning . . . 216
8.6 The theory of planning with sampled models ●● . . . 220
8.7 From tables to deep models ●●● . . . 221
8.8 Summary . . . 222
8.9 Further Reading . . . 223
8.10 Exercises . . . 223

Part III Generalization

9 Value Function Approximation 229
9.1 Why approximate 230
9.2 Features. 231
9.3 Gradient Monte Carlo and semi-gradient TD 235
9.4 The TD fixed point 238
9.5 Convergence and error of linear TD ●● 240
9.6 Least-squares TD 243
9.7 Control with function approximation 244
9.8 The deadly triad 246
9.9 Gradient-TD methods ●● 248
9.10 Fitted value iteration and fitted Q iteration 251
9.11 Theory of reinforcement learning with function approximation ●●● 254
9.12 Summary 255
9.13 Further Reading 255
9.14 Exercises 256

10 Deep Value-Based Reinforcement Learning 261
10.1 From fitted Q iteration to deep Q-networks 262
10.2 The DQN algorithm 262
10.3 DQN on CartPole, step by step 266
10.4 Overestimation and Double DQN 268
10.5 Dueling networks 270
10.6 Prioritized experience replay 271
10.7 Multi-step targets, noisy networks and Rainbow. 272
10.8 Distributional reinforcement learning ●● 273
10.9 Scaling up ●●● 277
10.10 Practice 278
10.11 Summary 279
10.12 Further Reading 280
10.13 Exercises 280

Part IV Policy Optimization

11 Policy Gradient Methods 287
11.1 Why optimize the policy directly 288
11.2 Policy parameterizations 289
11.3 The policy gradient theorem 293
11.4 REINFORCE 296
11.5 Baselines 298
11.6 Actor-critic methods 300
11.7 Generalized advantage estimation 304
11.8 Practical policy gradients 306
11.9 Subtleties: discounting, average reward and gradient estimators ●● 308
11.10 Compatible function approximation and natural actor-critic ●●● 310
11.11 Summary 311
11.12 Further Reading 312
11.13 Exercises 312

12 Natural Gradients, Trust Regions, and PPO . . . 317
12.1 The step-size problem . . . 318
12.2 Surrogate objectives . . . 319
12.3 The monotonic improvement bound ●● . . . 321
12.4 Natural policy gradient . . . 323
12.5 Trust region policy optimization . . . 326
12.6 Proximal policy optimization . . . 328
12.7 PPO in practice . . . 331
12.8 What PPO really optimizes ●●● . . . 334
12.9 Summary . . . 335
12.10 Further Reading . . . 336
12.11 Exercises . . . 336

13 Off-Policy Actor-Critic for Continuous Control . . . 341
13.1 Continuous actions . . . 342
13.2 The deterministic policy gradient . . . 343
13.3 Deep deterministic policy gradient (DDPG) . . . 346
13.4 Twin delayed DDPG (TD3) . . . 348
13.5 Maximum-entropy reinforcement learning . . . 349
13.6 Soft actor-critic (SAC) . . . 354
13.7 Pendulum with DDPG, TD3 and SAC . . . 358
13.8 Comparing the families . . . 360
13.9 Further developments ●● . . . 361
13.10 Summary . . . 363
13.11 Further Reading . . . 364
13.12 Exercises . . . 364

14 The Theory of Policy Optimization . . . 369
14.1 The landscape of the objective . . . 370
14.2 Gradient domination ●● . . . 372
14.3 Softmax policy gradient ●● . . . 375
14.4 Natural policy gradient and policy mirror descent ●● . . . 377
14.5 Entropy regularization ●● . . . 381
14.6 Function approximation ●●● . . . 383
14.7 Beyond tabular MDPs ●●● . . . 384
14.8 Summary . . . 385
14.9 Further Reading . . . 385
14.10 Exercises . . . 386

Part V Models, Exploration, and Data

15 Model-Based Reinforcement Learning . . . 393
15.1 What a model buys and what it costs . . . 394
15.2 Learning models . . . 397
15.3 The simulation lemma ●● . . . 400
15.4 Planning at every step: model-predictive control . . . 403
15.5 Learning a policy from model rollouts: MBPO . . . 405
15.6 Latent world models . . . 408
15.7 Learned models in games: AlphaZero and MuZero . . . 410
15.8 How much data does a model need? ●●● . . . 412

15.9 Summary . . . 413
15.10 Further Reading . . . 414
15.11 Exercises . . . 414

16 Exploration . . . 419
16.1 Why exploration is hard in MDPs . . . 420
16.2 Optimism in the face of uncertainty . . . 422
16.3 Posterior sampling . . . 426
16.4 Measuring exploration ●● . . . 428
16.5 Regret bounds ●● . . . 429
16.6 Exploration with function approximation ●●● . . . 432
16.7 Intrinsic motivation for deep reinforcement learning . . . 433
16.8 Exploration in practice . . . 436
16.9 Summary . . . 437
16.10 Further Reading . . . 438
16.11 Exercises . . . 438

17 Offline Reinforcement Learning and Off-Policy Evaluation . . . 443
17.1 Learning from a fixed data set . . . 444
17.2 Why off-policy learning fails offline . . . 445
17.3 Off-policy evaluation . . . 448
17.4 Keeping the policy close to the data . . . 456
17.5 Conservative and in-sample methods . . . 458
17.6 Model-based offline RL ●● . . . 463
17.7 The theory of offline RL ●● . . . 464
17.8 Offline RL as sequence modeling ●● . . . 467
17.9 Offline RL in practice . . . 468
17.10 Summary . . . 469
17.11 Further Reading . . . 470
17.12 Exercises . . . 470

18 Imitation Learning and Inverse Reinforcement Learning . . . 475
18.1 Learning from demonstrations . . . 476
18.2 Behavior cloning . . . 477
18.3 Interactive imitation: DAgger . . . 480
18.4 Inverse reinforcement learning . . . 483
18.5 Maximum-entropy IRL . . . 484
18.6 Adversarial imitation . . . 486
18.7 Learning rewards from preferences . . . 488
18.8 Modern imitation ●● . . . 490
18.9 Summary . . . 491
18.10 Further Reading . . . 492
18.11 Exercises . . . 493

Part VI Beyond a Single Agent and a Known State

19 Partial Observability and Memory . . . 499
19.1 When the state is hidden . . . 500
19.2 Beliefs . . . 501
19.3 Value functions over beliefs ●● . . . 504

19.4 Approximate planning ●● . . . 506
19.5 Learning without a model . . . 508
19.6 Information states and representations ●●● . . . 512
19.7 Summary . . . 514
19.8 Further Reading . . . 514
19.9 Exercises . . . 515

20 Hierarchy, Goals, and Temporal Abstraction . . . 519
20.1 Why structure over time helps . . . 520
20.2 Options and semi-Markov decision processes . . . 522
20.3 Where options come from . . . 529
20.4 Feudal and goal-conditioned hierarchies . . . 533
20.5 Goal-conditioned reinforcement learning . . . 534
20.6 The successor representation and successor features ●● . . . 537
20.7 What hierarchy has and has not delivered ●●● . . . 541
20.8 Summary . . . 542
20.9 Further Reading . . . 543
20.10 Exercises . . . 543

21 Multi-Agent Reinforcement Learning and Games . . . 547
21.1 Many learners . . . 548
21.2 Game theory essentials . . . 549
21.3 Learning in zero-sum games . . . 554
21.4 Independent learners and teams . . . 558
21.5 Centralized training with decentralized execution . . . 559
21.6 Imperfect-information games ●● . . . 561
21.7 Theory ●●● . . . 565
21.8 Landmarks and frontiers ●●● . . . 567
21.9 Summary . . . 568
21.10 Further Reading . . . 569
21.11 Exercises . . . 569

Part VII Reinforcement Learning in the World

22 Reinforcement Learning for Language Models . . . 575
22.1 Language models as policies . . . 576
22.2 The RLHF pipeline . . . 578
22.3 The KL-regularized optimum . . . 580
22.4 PPO for language models . . . 584
22.5 Direct preference optimization . . . 585
22.6 Reinforcement learning with verifiable rewards . . . 588
22.7 KL estimators ●● . . . 593
22.8 Reward hacking and overoptimization . . . 595
22.9 Process versus outcome rewards ●● . . . 598
22.10 Open problems ●●● . . . 599
22.11 Summary . . . 600
22.12 Further Reading . . . 601
22.13 Exercises . . . 602

23 Doing RL Well: Engineering, Evaluation, and Safety . . . 607
23.1 Why reinforcement learning is hard to get right . . . 608
23.2 Implementation . . . 609
23.3 Debugging . . . 612
23.4 Evaluation and statistics . . . 614
23.5 Hyperparameters . . . 620
23.6 Reward design . . . 622
23.7 Safe and constrained reinforcement learning . . . 625
23.8 From simulation to reality . . . 631
23.9 Compute and scale ●● . . . 632
23.10 Reproducibility and ethics . . . 633
23.11 Summary . . . 634
23.12 Further Reading . . . 634
23.13 Exercises . . . 635

24 Frontiers and Open Problems . . . 639
24.1 Where the field stands . . . 640
24.2 Generalization . . . 641
24.3 Foundation models and decision making . . . 644
24.4 Meta-RL and in-context RL . . . 645
24.5 Continual RL and plasticity . . . 646
24.6 Theory frontiers ●●● . . . 649
24.7 Reinforcement learning for science and engineering . . . 650
24.8 Open problems . . . 651
24.9 How to do RL research . . . 653
24.10 Summary . . . 654
24.11 Further Reading . . . 655
24.12 Exercises . . . 655

A Mathematical Toolkit . . . 659
A.1 Probability . . . 659
A.2 Markov chains . . . 662
A.3 Linear algebra . . . 662
A.4 Contraction mappings . . . 663
A.5 Optimization . . . 664
A.6 Information theory . . . 666
A.7 Exercises . . . 667

B Stochastic Approximation and Concentration . . . 669
B.1 Martingales . . . 669
B.2 The Robbins–Monro algorithm . . . 670
B.3 The ODE method . . . 673
B.4 Asynchronous stochastic approximation . . . 674
B.5 Two timescales . . . 675
B.6 Concentration inequalities . . . 676
B.7 Further reading . . . 680
B.8 Exercises . . . 680

C Neural Networks in a Nutshell . . . 683
C.1 Multilayer perceptrons . . . 683
C.2 Backpropagation . . . 684

C.3 Optimizers . . . 685
C.4 Initialization, normalization and regularization . . . 686
C.5 Convolutional and recurrent networks, and transformers . . . 687
C.6 Practical tips for reinforcement learning . . . 688
C.7 A walk through `rlcourse.nn` . . . 688
C.8 Exercises . . . 690

D The Lab Guide . . . 693
D.1 Setting up . . . 693
D.2 The `rlcourse` package . . . 694
D.3 The labs . . . 696
D.4 Running the tests . . . 697
D.5 Moving to real libraries . . . 697
D.6 Exercises . . . 699

References . . . 701

Preface

Reinforcement learning is the part of machine learning that is about decisions: how an agent should act when its actions have consequences that unfold over time, and when the only feedback it receives is a number that says how well things went. The field has grown from a corner of psychology and control theory into the technology behind game-playing programs that beat world champions, controllers for robots and fusion reactors, and the training of the language models that many people now use every day.

This book is a complete course on the subject. It starts from the beginning, with a three-state example you can solve on the back of an envelope, and ends at the research frontier, with the open problems of 2026. In between it covers the classical theory (Markov decision processes, dynamic programming, Monte Carlo and temporal-difference learning), the methods that made deep reinforcement learning work (DQN, policy gradients, PPO, SAC, model-based methods), learning from data and demonstrations, partial observability, hierarchy, multi-agent learning, and reinforcement learning for language models.

Who this book is for

The book is written for three kinds of readers at once, and every section is marked with the reader it is for.

- Foundations (●). Motivated newcomers: students, engineers and self-learners who know high-school algebra, a little probability and some Python. The ● sections alone form a complete introductory course: they explain the ideas, the core algorithms, and how to use and debug them, with worked numbers and small programs.
- Graduate (●●). Readers with calculus, linear algebra and probability, at the level of a first-year graduate course. The ●● sections add the derivations and proofs, and the modern algorithms in full detail.
- Research (●●●). Readers who want to do research. The ●●● sections cover sharp theory, current methods and open problems, and give an annotated path into the literature.

You do not have to decide in advance. Read the ● sections first; when a ●● section catches your interest, read it; skip the ones that do not. The ● sections never depend on the others.

What makes this book different

One running example. Almost every chapter returns to the *Trail*, a three-state problem introduced in Chapter 1. Because you already know its answer, you can see exactly what each new

method does: how Monte Carlo estimates converge, how temporal-difference learning propagates values, what a policy gradient looks like, why a trust region helps, what goes wrong offline.

Everything is computed. Every number in the text, every table and every figure comes from code in the book's labs, and the code is tested against exact answers. If the text says the uniform policy of the Trail is worth 1.76, a test checks it.

Labs you can read. Each chapter comes with short Python programs built on a small library, `rl-course`, that implements the book's algorithms in plain NumPy, including a tiny neural-network module, so that nothing is hidden behind a framework. The labs run on a laptop in minutes.

Exercises with solutions. Every chapter ends with exercises at all three levels, from by-hand calculations to proofs, programming tasks and research questions. The companion volume contains a complete solution to every one.

Honest about what is known. The book distinguishes what is proved from what is observed in experiments, and what is established from what is debated. Where the field does not yet know the answer, the book says so and points to the work in progress.

How the book was made

This edition replaces the earlier course notes of the Research_RL knowledge center. It was planned, written and checked with the help of AI tools, under a single style guide and outline. Every chapter was checked by automated tests of its numbers and code, and by builds that check every cross-reference and every exercise solution. Errors surely remain. If you find one, please record it so that the next edition can fix it.

The book stands on the shoulders of the classic texts, above all Sutton and Barto's *Reinforcement Learning: An Introduction*, and of the many researchers whose papers are cited throughout. The interactive companion course in the same knowledge center (`Apps_claude/RL Course`) lets you explore the Trail and the main algorithms in a web browser.

How to Use This Book

Three tracks

Every section of the book is marked with a reading level:

Table 1. The three reading tracks.

Mark	Track	What you get	What you need
●	Foundations	the ideas, the core algorithms, worked examples, practical advice	high-school algebra, basic probability (averages, conditional probability), some Python
●●	Graduate	derivations, proofs, the modern algorithms in full detail	calculus, linear algebra, probability at the level of a first university course
●●●	Research	sharp theory, recent methods, open problems	comfort with proofs; the ●● material

A section without a mark is a ● section. A section's subsections share its level unless they are marked higher. Exercises carry the same marks. In the web edition, the Track buttons at the top fold away every section above your track to a single line, which you can open with one click.

If your background is thin in places, the appendices help: Appendix A collects the probability, linear algebra, optimization and information theory the book uses; Appendix B the stochastic-approximation and concentration results behind the convergence proofs; Appendix C the essentials of neural networks; Appendix D explains the labs.

What to read first

Part I and Part II are the foundation for everything else. After them the book branches. Table 2 lists, for each chapter, the chapters you should have read first (their ● sections are enough unless a ●● section of the later chapter says otherwise).

Table 2. Chapter prerequisites.

Chapter	Read first
1 What Is Reinforcement Learning?	none
2 Bandits	1
3 Markov Decision Processes	1
4 Dynamic Programming	3
5 Monte Carlo Methods	4
6 Temporal-Difference Learning	5
7 Multi-Step Methods and Eligibility Traces	6
8 Planning and Learning with Models	6
9 Value Function Approximation	6
10 Deep Value-Based RL	9
11 Policy Gradient Methods	6 (2 helps)
12 Natural Gradients, Trust Regions, and PPO	11
13 Off-Policy Actor-Critic for Continuous Control	10, 12
14 The Theory of Policy Optimization	12
15 Model-Based RL	8, 10
16 Exploration	2, 6
17 Offline RL and Off-Policy Evaluation	5, 10 (13 helps)
18 Imitation Learning and Inverse RL	11
19 Partial Observability and Memory	6
20 Hierarchy, Goals, and Temporal Abstraction	6, 10
21 Multi-Agent RL and Games	4, 11
22 RL for Language Models	11, 12 (18 helps)
23 Doing RL Well	6, 11
24 Frontiers and Open Problems	as much of the book as you like

Suggested routes

A self-study introduction (about 12 weeks, ● sections). Week 1: Chapters 1–2. Week 2: Chapter 3. Week 3: Chapter 4. Week 4: Chapter 5. Weeks 5–6: Chapter 6. Week 7: Chapter 7 or 8. Week 8: Chapter 9. Week 9: Chapter 10. Week 10: Chapter 11. Week 11: Chapter 12. Week 12: Chapters 22 and 23. Do every lab, and the ● exercises.

A one-semester undergraduate course (14 weeks). The self-study route, adding Chapter 13 (● sections) and the ● sections of Chapter 17, with a programming project in the last month (for example, PPO or DQN on a problem the students choose, evaluated with the methods of Chapter 23).

A one-semester graduate course (14 weeks, ● and ●● sections). Week 1: Chapters 1–2 (including

the regret analysis). Week 2: Chapter 3. Week 3: Chapter 4. Week 4: Chapters 5–6. Week 5: Chapter 6 (convergence) and 7. Week 6: Chapter 9. Week 7: Chapter 10. Week 8: Chapter 11. Week 9: Chapter 12. Week 10: Chapter 13. Week 11: Chapter 14. Week 12: Chapters 15 or 16. Week 13: Chapters 17 and 18. Week 14: Chapter 22. Assign one ●● exercise and one lab each week, and a research-style project.

A deep-RL practitioner's track. Chapter 1; Chapter 3 (●); Chapter 6; Chapter 9 (●); Chapters 10–13; Chapter 23; then Chapters 15, 17 and 22 as your applications require.

Reinforcement learning for language models. Chapters 1, 2 and 3 (●); Chapter 6 (●); Chapters 11 and 12; Chapter 18 (the sections on preference learning); Chapter 22; Chapter 23.

A research reading group or second graduate course (●● and ●●●). Chapter 2 (lower bounds and beyond), Chapter 4 (complexity and linear programming), Chapter 9 (theory of function approximation), Chapter 14, Chapter 16, Chapter 17 (pessimism), Chapter 21 (theory), Chapter 22 (open problems), Chapter 24, with Appendix B as the toolkit. Read the papers in each chapter's Further Reading alongside.

Labs, exercises and solutions

The labs live in the `labs/` folder. Each is a Python script that also opens as a notebook (in VS Code, or in Jupyter with Jupytext). They use a small library, `rlcourse`, written in plain NumPy so that you can read every line. To run them you need Python 3.10 or later with NumPy and Matplotlib; `pytest` runs the tests. Appendix D explains the setup and the library, and how to move on to libraries such as Gymnasium, PyTorch, Stable-Baselines3 and CleanRL.

Every chapter ends with exercises. The companion volume, *Solutions to the Exercises*, contains a complete solution to each, with the exercise statement repeated so that it can be read on its own. In the web edition each exercise links to its solution. Try an exercise before reading its solution: the effort is where the learning happens.

Conventions

The notation is collected on the next pages and is the same throughout the book. Definitions, theorems, lemmas and propositions share one counter per chapter (Definition 3.1, Theorem 3.2, …); examples, algorithms, exercises, figures, tables and equations each have their own. The boxed notes have fixed roles:

Table 3. The boxed notes.

Box	Use
What you will learn	the goals of the chapter
Intuition	a plain-language explanation of a formal idea
Key idea	the one thing to remember from a section
Pitfall	a common mistake or misconception

Box	Use
Checkpoint	quick questions to test yourself, with answers right after
Lab	which program to run and what to look for
Historical note, Remark, Note	asides
Research note	open questions, in lower-level sections

Notation

The whole book uses one set of symbols. This page collects them. When a chapter needs a symbol only locally, it says so where the symbol first appears.

Conventions. Lowercase letters with a time index, such as s_t, a_t and r_t, are the random quantities produced by the agent and the environment. The same letters without an index, such as s and a, are particular values. Vectors are lowercase (w, θ, $\phi(s)$) and matrices are uppercase (P, Φ). We write $\Pr(\cdot)$ for probabilities and $\mathbb{E}[\cdot]$ for expectations; a subscript such as $\mathbb{E}_\pi$ says which policy generates the data.

The interaction

Symbol	Meaning
$t = 0, 1, 2, \ldots$	time step
$s_t \in \mathcal{S}$	state at time t; $\mathcal{S}$ is the state space
$a_t \in \mathcal{A}$	action at time t; $\mathcal{A}$ is the action space ($\mathcal{A}(s)$ if it depends on s)
r_t	reward received right after taking a_t in s_t
o_t	observation, when the agent does not see the state (Chapter 19)
T	final time step of an episode (a random variable)
H	horizon of a finite-horizon problem
$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$	a trajectory

With this convention a transition is (s_t, a_t, r_t, s_{t+1}). Sutton and Barto's textbook writes the same reward as R_{t+1}; only the index differs.

Markov decision processes

Symbol	Meaning
$P(s' \mid s, a)$	transition probability: probability of moving to s' after taking a in s
$r(s, a)$	expected reward $\mathbb{E}[r_t \mid s_t = s, a_t = a]$

Symbol	Meaning
$\gamma \in [0, 1)$	discount factor ($\gamma = 1$ is allowed only in episodic problems)
d_0	initial-state distribution: $s_0 \sim d_0$
$(\mathcal{S}, \mathcal{A}, P, r, \gamma, d_0)$	a Markov decision process (MDP)
$\lvert\mathcal{S}\rvert$, $\lvert\mathcal{A}\rvert$	numbers of states and actions in a finite MDP

Policies, returns and values

Symbol	Meaning
$\pi(a \mid s)$	stochastic policy: probability of taking a in s
$\pi(s)$, $\mu_\theta(s)$	deterministic policies (the action chosen in s)
π_θ	policy with parameters θ
$b(a \mid s)$	behavior policy, the one that generated the data in off-policy learning
$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$	return from time t (the sum stops at T in an episode)
$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$	state-value function of π
$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a]$	action-value function of π
$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$	advantage function of π
V^*, Q^*, π^*	optimal value functions and an optimal policy
$J(\pi) = \mathbb{E}_{s_0 \sim d_0}[V^\pi(s_0)]$	expected return of π, the objective; $J(\theta)$ for π_θ
$d^\pi(s) = (1 - \gamma) \sum_t \gamma^t \Pr_\pi(s_t = s)$	discounted state-visitation distribution
$\mathcal{T}^\pi$, $\mathcal{T}^*$	Bellman expectation and optimality operators
P^π, r^π	transition matrix and expected-reward vector when following π

Learning

Symbol	Meaning
η	step size (learning rate); η_t if it changes over time
ε	exploration rate of an ε-greedy policy
$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$	temporal-difference (TD) error
$G_{t:t+n}$	n-step return; G_t^λ is the λ-return
$\lambda \in [0, 1]$	trace-decay (TD(λ)) or GAE parameter
z_t	eligibility trace vector
$\rho_t = \pi(a_t \mid s_t)/b(a_t \mid s_t)$	importance-sampling ratio

Symbol	Meaning
$\phi(s), \phi(s,a) \in \mathbb{R}^d$	feature vector
w	parameters of a value function (critic), e.g. $V_w(s) = w^\top \phi(s)$
$\bar{w}$	parameters of a target network
θ	parameters of a policy (actor)
$\hat{A}_t$	advantage estimate used in a policy update
$\mathcal{D}$	a data set, or a replay buffer
$\mathcal{H}(\pi(\cdot \mid s))$	entropy of the action distribution in s
$D_{\mathrm{KL}}(p \,\|\, q)$	Kullback–Leibler divergence
α	entropy temperature (soft actor-critic)
ϵ	clipping range of PPO (different from the exploration rate ε)
τ	target-network averaging rate: $\bar{w} \leftarrow (1-\tau)\bar{w} + \tau w$
β	strength of a KL penalty (RLHF, DPO)
κ	expectile level in implicit Q-learning (Chapter 17)

Bandits (Chapter 2)

Symbol	Meaning
K	number of arms
μ_a	mean reward of arm a; $\mu^* = \max_a \mu_a$
$\Delta_a = \mu^* - \mu_a$	gap of arm a
$\hat{\mu}_a, N_a$	estimated mean and number of pulls of arm a so far
Reg_n	regret after n rounds

The arm mean μ_a is the action value $Q(a)$ of a problem with a single state; Chapter 2 uses the bandit literature's symbol.

Language models (Chapter 22)

Symbol	Meaning
x	prompt
$y = (y_1, \dots, y_L)$	response, a sequence of tokens
$\pi_\theta(y \mid x)$	the language model viewed as a policy
π_{ref}	reference model (usually the supervised fine-tuned model)
$r_\psi(x, y)$	learned reward model with parameters ψ

The Trail

The running example appears in almost every chapter. Its states are Start, Middle and Goal, and the episode ends in End. In Start and Middle the actions are go (move on, reward 0) and exit (end the episode, reward +1 in Start and +2 in Middle). In Goal the only action, finish, ends the episode with reward +4. A policy is described by two numbers,

$$p_0 = \pi(\text{go} \mid \text{Start}), \qquad p_1 = \pi(\text{go} \mid \text{Middle}),$$

and unless a chapter says otherwise $\gamma = 0.9$. Two values are worth remembering. The uniform policy ($p_0 = p_1 = \frac{1}{2}$) has $V^\pi(\text{Start}) = 1.76$ and $V^\pi(\text{Middle}) = 2.8$. The optimal policy goes all the way and has $V^*(\text{Start}) = 3.24$ and $V^*(\text{Middle}) = 3.6$.

Reading levels

Mark	Track	Meaning
(none) or ●	Foundations	core material for every reader
●●	Graduate	derivations, proofs and advanced methods
●●●	Research	sharp theory, recent work and open problems

Part I

Foundations

What is a reinforcement-learning problem, and what does it mean to solve one? Part I answers both questions. Chapter 1 introduces the agent, the environment and the reward through the Trail, the small example that runs through the whole book. Chapter 2 studies the simplest case, a single repeated choice, where the tension between exploring and exploiting can be analyzed exactly. Chapter 3 defines Markov decision processes, values and the Bellman equations, the language of everything that follows, and Chapter 4 shows how to solve them exactly when the model is known.

CHAPTER 1

What Is Reinforcement Learning?

WHAT YOU WILL LEARN

- Describe a reinforcement-learning problem in terms of an agent, an environment, states, actions and rewards.
- Compute discounted returns and the expected return of a simple stochastic policy by hand.
- Explain the four difficulties that make reinforcement learning hard: delayed consequences, exploration, generalization, and data that depends on the learner's own behavior.
- Place reinforcement learning among the other kinds of machine learning and among its neighboring fields.
- Name the main families of reinforcement-learning methods and say where each is covered in this book.
- Show that the best policy of a small decision problem is deterministic, and preview why that holds in general (●●).
- Discuss what rewards can and cannot express (●●●).

A child learning to ride a bicycle is not told which muscles to move. Nobody hands her a table of the correct handlebar angle for every speed and every lean. She tries something, wobbles, falls or does not, and gradually her behavior changes so that falling becomes rare. What drives the change is not instruction but consequences: some ways of acting turn out well and others turn out badly.

Reinforcement learning (RL) is the study of this kind of learning in machines: learning what to do, by trial and error, so as to maximize a numerical signal that measures how well things are going. It is the branch of machine learning concerned with decisions whose effects unfold over time. The same framework describes a program that teaches itself to play Go, a controller that keeps a plasma stable inside a fusion reactor, a robot that learns to walk, and a language model that is trained to solve mathematics problems by being rewarded for correct answers.

This chapter builds the vocabulary and the intuition for the rest of the book. We start with the basic loop between an agent and its environment (Section 1.2), then meet the *Trail*, a three-state problem that we will use as a running example in almost every chapter (Section 1.3). With the Trail in hand we can see concretely what makes reinforcement learning hard (Section 1.4), how it differs from other kinds of learning (Section 1.5), and what an agent is made of (Section 1.6). Section 1.7 runs a first experiment. Section 1.8 (●●) proves a small but important fact about the Trail, and Section 1.9 (●●●) asks what rewards can and cannot express. A short history (Section 1.10) and a map of the book (Section 1.11) close the chapter. If you are new to the subject, read Section 1.1 to 1.7, 1.10 and 1.11 and come back to the rest later.

1.1 Learning from consequences

Much of what animals learn, they learn from the consequences of their actions. Around 1900 the psychologist Edward Thorndike put cats in "puzzle boxes" that opened when the cat pressed a lever or pulled a loop of string. At first the cats struggled at random; over many trials they escaped faster and faster. Thorndike summarized the pattern as the *law of effect*: actions followed by satisfaction become more likely in the same situation, and actions followed by discomfort become less likely (Thorndike 1911). Animal trainers use the same principle every day, and it is the root of the word *reinforcement*.

Reinforcement learning turns this principle into a precise computational problem. Consider four very different situations:

- A game-playing program chooses moves in chess or Go. The only feedback is whether it eventually wins or loses, often hundreds of moves after the decisions that mattered.
- A robot learns to walk. At each instant it chooses torques for its motors; it is rewarded for moving forward and penalized for falling or for using too much energy.
- A recommendation system chooses which article to show a reader. A click is a small reward, but showing only sensational articles may earn clicks today and lose readers next month.
- A language model writes a solution to a mathematics problem, one token at a time. At the end, an automatic checker marks the answer as right or wrong, and that single bit of feedback is used to improve the model.

These problems share a structure. A decision maker acts, the world responds, and a number tells the decision maker how good the outcome was. The number does not say what the right action would have been; it only grades what actually happened. And the consequences of an action may appear long after it was taken.

> **KEY IDEA**
>
> Reinforcement learning is learning to act from *evaluative* feedback: a reward that grades the outcomes of your own choices, often with a delay, rather than *instructive* feedback that tells you the correct choice.

1.2 The agent–environment loop

Every reinforcement-learning problem has the same basic shape, shown in Figure 1.1. The learner and decision maker is called the *agent*. Everything it interacts with, everything outside its direct control, is the *environment*. They interact in a loop that runs over discrete time steps $t = 0, 1, 2, \ldots$:

1. The agent observes the current *state* s_t of the environment: a description of the situation, such as the positions of the pieces on a board or the joint angles and velocities of a robot.
2. It chooses an *action* a_t: a move, a set of motor torques, an article to show.
3. The environment responds with a numerical *reward* r_t and moves to a new state s_{t+1}.

Then the loop repeats from s_{t+1}.

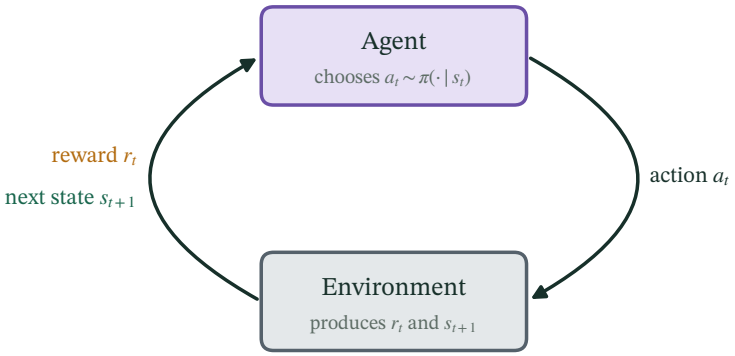


Figure 1.1. The agent–environment loop. At each time step the agent sees the state s_t and chooses an action a_t; the environment answers with a reward r_t and the next state s_{t+1}. The loop then repeats.

A few words about these ingredients, which later chapters make precise.

States. The state is whatever information the agent uses to decide. In this book's first two thirds we assume the agent sees a state that summarizes everything relevant about the past, so that the future depends on the past only through the present state. This is the *Markov property*, the subject of Chapter 3. When the agent sees only partial or noisy observations, as a poker player who cannot see the other players' cards, the problem is *partially observable* (Chapter 19).

Actions. Actions can be discrete (left or right, one of 361 points on a Go board) or continuous (a torque between -2 and 2 newton-meters). The set of available actions may depend on the state.

Rewards. The reward r_t is a single number received right after the action a_t. It is the only channel through which we tell the agent what we want. The reward for a robot might be its forward speed minus a penalty for energy; for a game it might be $+1$ for a win, -1 for a loss and 0 for every other move. The reward defines the goal, but it should not say *how* to reach the goal: rewarding a chess program for capturing pieces may teach it to capture pieces rather than to win.

Policy. The agent's way of behaving is its *policy*: a rule that says which action to take in each state. A policy may be deterministic (always the same action in a given state) or stochastic (a probability distribution over actions, written $\pi(a \mid s)$: the probability of taking action a in state s). Learning, in reinforcement learning, means changing the policy.

Episodes and returns. Many tasks break naturally into *episodes*: a game, a maze run, a conversation. An episode ends in a *terminal state*, and the next one starts afresh. Other tasks, like controlling a power grid, go on without end. Either way, the agent's goal is not to maximize the immediate reward but the total reward over the long run. The simplest version adds up rewards, but it is usually better to count rewards that arrive later a little less than rewards that arrive now. The *discounted return* from time t is

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{1.1}$$

where the *discount factor* γ is a number between 0 and 1. In words: a reward received k steps in the future is worth γ^k times what it would be worth now. In an episode that ends at time T, the sum simply stops at the last reward r_{T-1}.

Discounting has several justifications, which Chapter 3 discusses. It keeps the sum finite in tasks that never end. It expresses a preference for sooner rewards, much like interest rates in economics. And it encodes uncertainty: if there is a chance $1-\gamma$ at each step that the world will end (or the robot's battery will die), then a reward k steps away will be collected only with probability γ^k. A discount close to 1 makes the agent farsighted; a small discount makes it impatient.

The goal of reinforcement learning can now be stated in one sentence: find a policy that makes the expected return as large as possible.

> **PITFALL**
>
> The reward is not the return. The reward r_t is the immediate feedback for one action; the return G_t adds up all future rewards. An agent that maximizes immediate reward is greedy, and greed is usually a poor strategy, as the next section shows.

1.3 A first example: the Trail

To make these ideas concrete we use a tiny problem that will accompany us through the whole book. We call it the *Trail* (Figure 1.2).

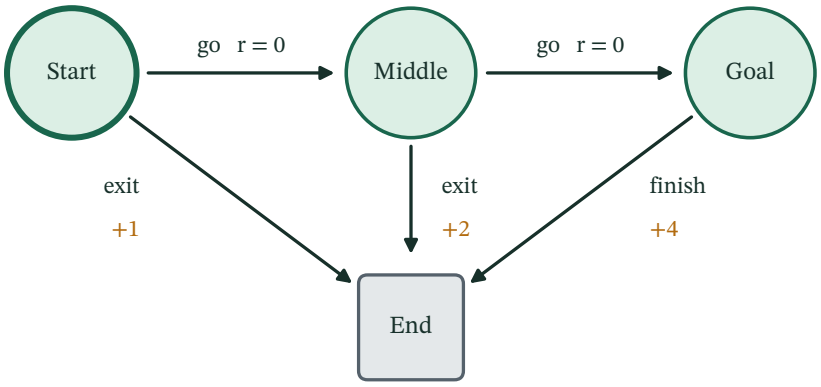


Figure 1.2. The Trail, the running example of this book. A hiker starts in Start. In Start and Middle she can go on (no reward yet) or exit (a reward of +1 in Start or +2 in Middle, and the trip is over). In Goal the only action is finish, worth +4. Every path ends in End.

A hiker sets out on a trail with three stations: Start, Middle and Goal. At Start and at Middle she has two choices. She can exit the trail, collecting a reward on the way out (+1 at Start, +2 at Middle), and the trip is over. Or she can go on to the next station, which earns nothing immediately. At Goal there is only one thing to do, finish, which earns +4 and also ends the trip. The end of the trip is a terminal state, called End.

The Trail is small enough to solve by hand, yet it already has the essential tension of reinforcement learning: the best rewards are the delayed ones, and getting them requires passing up smaller rewards along the way.

1.3.1 Three paths and their returns

Every episode of the Trail follows one of three paths:

Table 1.1. The three possible episodes of the Trail and their returns, from (1.1).

Path	Rewards	Discounted return G_0	With $\gamma = 0.9$
exit at Start	1	1	1.00
go, then exit at Middle	0, 2	$0 + \gamma \cdot 2 = 2\gamma$	1.80
go, go, finish	0, 0, 4	$0 + \gamma \cdot 0 + \gamma^2 \cdot 4 = 4\gamma^2$	3.24

With $\gamma = 0.9$, going all the way is clearly best: 3.24 beats 1.80 and 1. But the answer depends on how patient the hiker is. Figure 1.3 plots the three returns as functions of γ. Below $\gamma = 0.5$ the immediate +1 is worth more than the discounted later rewards; above 0.5 the full trip wins. At exactly $\gamma = 0.5$ all three paths are worth exactly 1.

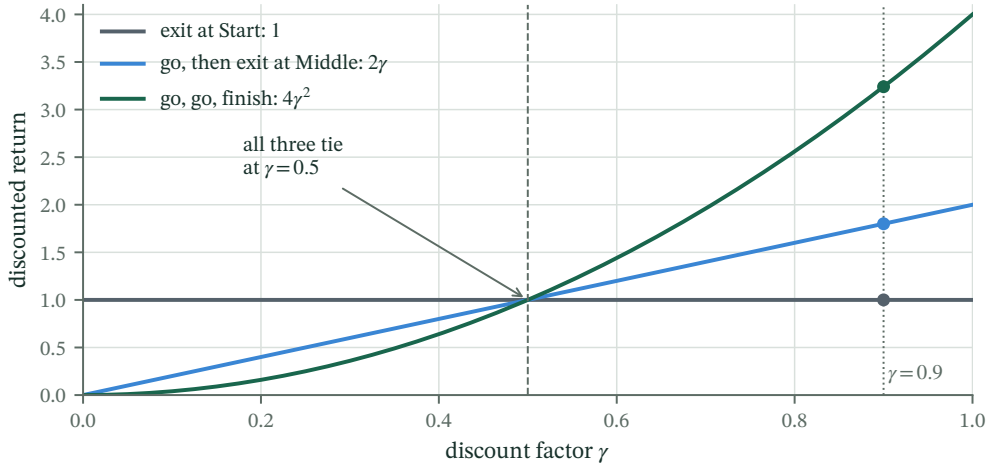


Figure 1.3. Discounted returns of the Trail's three paths as functions of the discount factor. Patience pays: for $\gamma > 0.5$ the longest path is best, for $\gamma < 0.5$ the immediate exit is best, and at $\gamma = 0.5$ all three tie. The dots mark the default $\gamma = 0.9$.

This small calculation already teaches two lessons. First, an agent that maximizes the *immediate* reward exits at Start and collects 1, the worst outcome when $\gamma = 0.9$. Good behavior requires thinking about the long run. Second, the discount factor is part of the problem definition: it changes which behavior is best. Choosing γ is a modeling decision, and Chapter 3 discusses how to make it.

1.3.2 Policies and expected returns

A policy for the Trail must say what to do at Start and at Middle. A stochastic policy is described by two probabilities:

$$p_0 = \pi(\text{go} \mid \text{Start}), \qquad p_1 = \pi(\text{go} \mid \text{Middle}).$$

With probability $1 - p_0$ the hiker exits at Start; with probability p_0 she goes on, and then at Middle she goes on with probability p_1. Deterministic policies are the special cases where p_0 and p_1 are 0 or 1.

When the policy is stochastic, the return is a random variable, and the natural measure of how good the policy is is its *expected* return. For the Trail it is easy to compute, because we can list the paths, their probabilities and their returns.

Example 1.1 The expected return of the uniform policy

Take the policy that flips a fair coin at each decision: $p_0 = p_1 = \frac{1}{2}$. The three paths have probabilities

$$\Pr(\text{exit at Start}) = 1 - p_0 = \tfrac{1}{2}, \quad \Pr(\text{go, exit}) = p_0(1 - p_1) = \tfrac{1}{4}, \quad \Pr(\text{go, go, finish}) = p_0 p_1 = \tfrac{1}{4}.$$

With $\gamma = 0.9$ the returns are 1, 1.8 and 3.24, so the expected return is

$$\tfrac{1}{2} \cdot 1 + \tfrac{1}{4} \cdot 1.8 + \tfrac{1}{4} \cdot 3.24 = 0.5 + 0.45 + 0.81 = 1.76.$$

The same computation with general p_0, p_1 and γ gives the expected return of every policy:

$$J(p_0, p_1) = (1 - p_0) \cdot 1 + p_0(1 - p_1) \cdot 2\gamma + p_0 p_1 \cdot 4\gamma^2. \tag{1.2}$$

For the uniform policy this is 1.76; for the policy that always goes ($p_0 = p_1 = 1$) it is $4\gamma^2 = 3.24$. ♦

The expected return starting from a state, when following a policy π, is called the *value* of that state under π and is written $V^\pi(s)$. So Example 1.1 computed $V^\pi(\text{Start}) = 1.76$ for the uniform policy. Values are the central objects of reinforcement learning. In this example we computed the value by listing every possible path, which works only because the Trail has three. Chapter 3 shows how to compute values for any problem with a known model, using the *Bellman equations*, and most of the book is about estimating them from experience when the model is unknown.

CHECKPOINT

1. What is the return of the path go, exit when $\gamma = 0.5$?
2. Why is the return of exiting at Start equal to 1 for every γ?
3. What is the expected return of a policy with $p_0 = 0$?

Answers.

1. $2\gamma = 1$.
2. The reward arrives at time $t = 0$, and rewards at time 0 are not discounted ($\gamma^0 = 1$).
3. It is 1: a policy that never goes always exits at Start, whatever it would have done at Middle.

1.3.3 What the agent does not know

So far we have computed everything from a complete description of the Trail. A reinforcement-learning agent usually has no such description. It does not know in advance that finish pays 4, or even that Goal exists. It can only act and observe what happens. To discover that going on is worthwhile, it must try going on, and it must do so often enough to be confident that the delayed reward is real and not luck. This is where learning comes in, and Section 1.7 runs a first experiment.

1.4 What makes reinforcement learning hard

The Trail is easy. Real problems are not, and the reasons are worth naming now, because the rest of the book is organized around them.

Delayed consequences and credit assignment. In the Trail, the action go at Start earns nothing, yet it is the action that makes the +4 possible. When a reward finally arrives, the agent must work out which of its earlier actions deserve the credit. This is the *credit assignment problem*, named by Marvin Minsky in 1961 (Minsky 1961). In chess the decisive move may come thirty moves before the win; in a language model, a single wrong step early in a long derivation spoils the final answer. Much of the theory of reinforcement learning, from value functions (Chapter 3) to temporal-difference learning (Chapter 6) and eligibility traces (Chapter 7), is about assigning credit across time.

Exploration versus exploitation. An agent that has found something that works, such as exiting at Start for a sure +1, faces a dilemma. It can *exploit* what it knows and collect the +1, or *explore* the unknown action in the hope of something better, at the risk of doing worse in the meantime. An agent that never explores can never discover the +4 at Goal; an agent that always explores never profits from what it has learned. Balancing the two is a problem with no counterpart in supervised learning. Chapter 2 studies it in its simplest form, and Chapter 16 in full.

Generalization and scale. The Trail has three states. A game of Go has more than 10^{170} legal positions; a robot's state is a vector of continuous joint angles and velocities; a language model's state is the text so far. The agent will never see most states twice, so it must *generalize* from the states it has seen to similar ones it has not. Combining generalization with the other difficulties creates new problems of its own; Part III is devoted to them.

The data depends on the learner. In supervised learning the data set is fixed and the learner only fits it. In reinforcement learning the agent's policy determines which states it visits and which actions it tries, so it determines its own data. As the policy changes, the data change. This feedback loop can make learning unstable, and it means that bad early decisions can starve the agent of the information it needs to improve.

Two further difficulties appear in practice. *Evaluation is noisy*: returns vary from episode to episode, so telling a good policy from a lucky one takes many trials (Chapter 23). And *the reward must be specified*: writing down a reward that captures what we actually want is harder than it looks, and agents are very good at exploiting loopholes (Section 1.9, Chapters 22 and 23).

INTUITION

A useful way to keep these difficulties apart: credit assignment is about *time* (which past action caused this reward?), exploration is about *information* (what have I not tried?), generalization is about *space* (what do states I have not seen look like?), and the feedback loop is about *data* (my behavior decides what I learn).

1.5 Reinforcement learning among the learning problems

Machine learning is often divided into three kinds of problems, distinguished by the feedback the learner receives (Table 1.2).

Table 1.2. Three kinds of learning problems.

	Supervised learning	Unsupervised and self-supervised learning	Reinforcement learning
Feedback	the correct output for each input	none beyond the data	a reward that grades outcomes
Data	a fixed set of input–output pairs	a fixed set of inputs	generated by the agent's own actions
Goal	predict outputs for new inputs	find structure, compress, generate	maximize long-run reward
Central difficulty	generalization	defining a useful objective	credit assignment and exploration

In *supervised learning*, a teacher provides the correct answer for every example: this image shows a cat, this email is spam. The learner's task is to generalize these answers to new examples. If reinforcement learning were supervised, the teacher would say "at Start, the correct action is go". Instead, the reinforcement-learning agent only hears "that episode earned 1.8" and must work out for itself whether a different choice would have earned more.

In *unsupervised* and *self-supervised learning*, there is no feedback at all beyond the data; the goal is to find structure in it, to compress it, or to predict one part of it from another, as a language model predicts the next word. Such models are increasingly used inside reinforcement-learning agents, to represent states or to provide the initial policy, but finding structure is not the same as acting well.

Reinforcement learning also has close neighbors outside machine learning:

- Optimal control studies how to steer a dynamical system to minimize a cost, typically with a known model and continuous states. Its central tool, dynamic programming, is also ours (Chapter 4), and the boundary between "control" and "reinforcement learning" is largely one of tradition: control theory usually assumes a model, reinforcement learning usually does not.

- Operations research introduced Markov decision processes and studies them for inventory management, scheduling and queueing (Puterman 1994).
- Economics and game theory study decisions over time and decisions among several interacting agents (Chapter 21).
- Psychology and neuroscience gave reinforcement learning its name and some of its ideas, and in return use its algorithms as models of learning in the brain: the activity of dopamine neurons closely resembles the temporal-difference error of Chapter 6 (Schultz et al. 1997).

One more distinction will recur. In *planning*, the agent knows how the environment works (it has a *model*) and computes a good policy without acting; this is how we solved the Trail in Section 1.3. In *learning*, the agent must discover how the environment works, or at least how to act in it, from experience. Many methods mix the two: they learn a model from experience and plan with it (Chapters 8 and 15).

1.6 Inside an agent: policies, values and models

An agent can contain up to three kinds of learned components.

- A policy maps states to actions (or to probabilities of actions). Every agent has one, explicitly or implicitly; it is what the agent does.
- A value function predicts the expected return from each state, $V^\pi(s)$, or from each state and action, written $Q^\pi(s, a)$. Values are predictions that make credit assignment possible: a state with a high value is a good place to be, even if it earned no reward yet. In the Trail, a value function would tell the agent that Middle is worth more than the +2 available there, because it leads to Goal.
- A model predicts how the environment responds: the next state and reward for each state and action. With a model, the agent can plan by imagining the consequences of its actions before taking them.

Different combinations of these components give the main families of methods, which organize the book:

Table 1.3. The main families of reinforcement-learning methods and where to find them.

Family	What it learns	How it acts	Chapters
Value-based	a value function Q	picks the action with the highest estimated value	5–7, 9–10
Policy-based	a policy directly	samples from the policy	2, 11–12, 14
Actor-critic	a policy (actor) and a value function (critic)	the actor acts; the critic judges and guides it	11–13
Model-based	a model, plus values or a policy	plans with the model	8, 15

A few more distinctions describe how an agent learns rather than what it learns. A method is

on-policy if it learns about the policy it is currently following, and *off-policy* if it can learn about one policy (say, the best one) from data generated by another (say, an exploratory one). An agent learns *online* if it learns while interacting, and *offline* if it learns from a fixed data set collected earlier (Chapter 17). Finally, it helps to separate two tasks: *prediction*, or evaluating a given policy, and *control*, or finding a good policy. Prediction is usually the first step: it is easier to improve a policy once you can tell how good it is.

1.7 Learning by trial and error: a first experiment

Let us now forget the description of the Trail and behave like an agent that knows nothing but what it experiences. Two questions come first: how can it *evaluate* a policy, and how can it *improve* one?

Evaluating by averaging. The value $V^{\pi}(\text{Start})$ is an expected return, and the natural way to estimate an expectation is to average samples. Run the policy for many episodes, record the return of each, and average them. Figure 1.4 shows twenty independent runs of this experiment for the uniform policy. Each estimate wanders at first, because a single episode returns 1, 1.8 or 3.24, never 1.76; as episodes accumulate, every run settles near the true value. The shaded band shows the typical size of the error, which shrinks in proportion to $1/\sqrt{n}$ after n episodes: a hundred times more episodes buy ten times more accuracy. This method, estimating values by averaging complete returns, is the Monte Carlo method of Chapter 5.

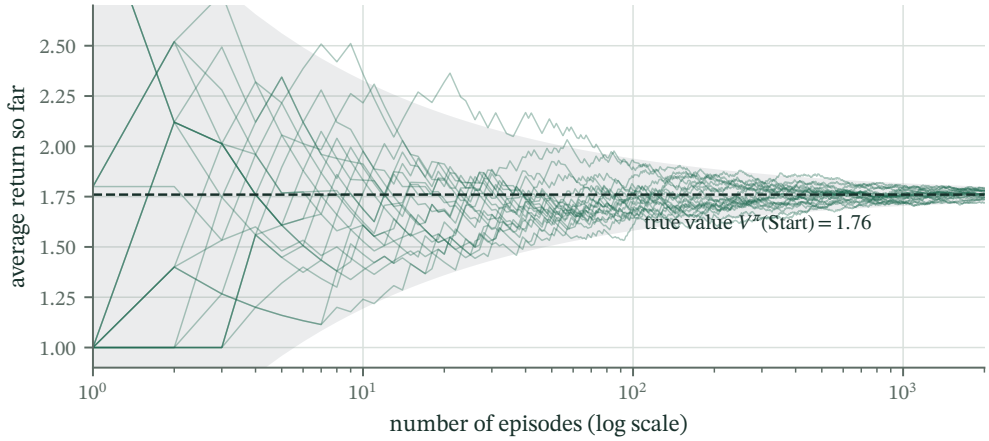


Figure 1.4. Estimating the value of the uniform policy by averaging returns. Each thin line is one run of the experiment, showing the average of the first n episodes. All runs converge to the true value 1.76 (dashed); the shaded band is the 95% range $1.76 \pm 1.96\,\sigma/\sqrt{n}$, where $\sigma \approx 0.91$ is the standard deviation of a single return.

Improving by trying. Suppose the agent must decide only what to do at Start, and that if it goes on, it keeps going. It can keep a running average of the returns that followed each choice, and choose the one with the higher average. Here is the catch: both averages start at zero. If the agent always chooses the better-looking option, it picks one arbitrarily (say exit), earns +1, and from then on exit looks better than the untried go, whose estimate is still zero. It never tries go again and never learns that it is worth 3.24. If instead the agent picks a random choice 10% of the time, it soon tries go, sees the larger return, and switches. In the lab, this agent chooses go

in about 95% of the last hundred episodes (the remaining 5% are its random exploratory choices). This small experiment contains the exploration–exploitation dilemma in its purest form, and the ε-greedy rule we used ("explore with probability ε") is the first algorithm of Chapter 2.

LAB

Lab 1.1: Meet the Trail (`labs/ch01_intro.py`, about 30 minutes). Run `python3 labs/ch01_intro.py`. You will play episodes of the Trail, estimate values by averaging returns and watch the error shrink, search for the best policy, find the discount factor at which the best policy changes, and build the exploring agent described above. The "Going further" ideas at the end make the Trail slippery.

TRY IT INTERACTIVELY

The interactive course in `Apps_claude/RL Course/index.html` (Module 1) lets you change p_0, p_1 and γ with sliders and watch the returns and values of the Trail change as you drag.

1.8 Why the best policy of the Trail is deterministic ●●

The Trail allows infinitely many stochastic policies, one for each pair (p_0, p_1) in the unit square. Yet the best policy we found is deterministic: go, and go again. This is not a coincidence, and the reason is worth seeing early, because the same idea, in a more general form, underlies the dynamic-programming methods of Chapters 3 and 4.

Figure 1.5 shows the expected return $J(p_0, p_1)$ of (1.2) over the whole square of policies. The maximum is in a corner.

Proposition 1.1 A deterministic policy is optimal for the Trail

For every $\gamma \in [0, 1]$, the function $J(p_0, p_1)$ of (1.2) attains its maximum over $[0, 1]^2$ at a corner, that is, at a deterministic policy.

Proof. Rewrite (1.2) as

$$J(p_0, p_1) = 1 + p_0\big[\, 2\gamma - 1 + p_1(4\gamma^2 - 2\gamma)\,\big].$$

For fixed p_1, J is an affine function of p_0, so on the interval $[0, 1]$ it is maximized at an endpoint, $p_0 = 0$ or $p_0 = 1$. If $p_0 = 0$, then $J = 1$ whatever p_1 is. If $p_0 = 1$, then $J = 2\gamma + p_1(4\gamma^2 - 2\gamma)$ is affine in p_1 and is maximized at $p_1 = 0$ or $p_1 = 1$. In every case some corner does at least as well as any interior point. ■

The proof is more interesting than its conclusion, because it reveals a way of *computing* the best policy: decide the last choice first. At Middle, the hiker compares exiting (2) with going on to Goal (worth 4, discounted to 4γ from the point of view of Middle). The better of the two is the value of being at Middle when acting optimally from there on:

$$V^*(\text{Middle}) = \max\{2,\ \gamma \cdot 4\}.$$

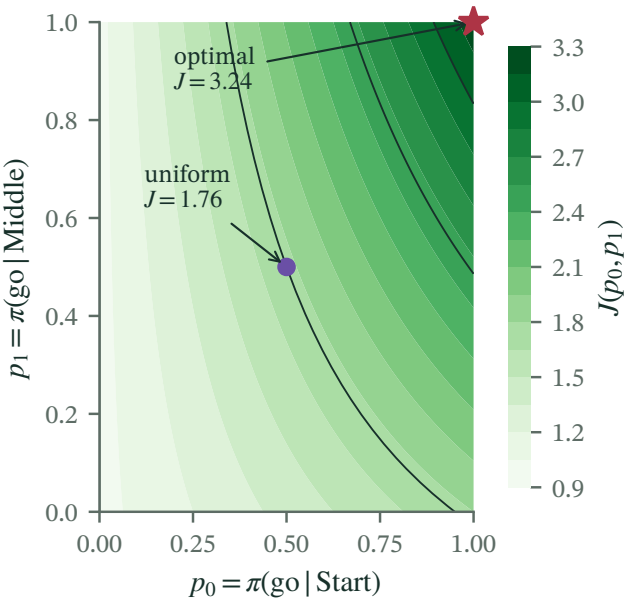


Figure 1.5. The expected return $J(p_0, p_1)$ of every policy of the Trail, for $\gamma = 0.9$. Darker is better. The uniform policy (dot) earns 1.76; the best policy is the corner $p_0 = p_1 = 1$ (star), which earns 3.24. The contour lines are curved, but along every horizontal or vertical line J changes linearly.

At Start she compares exiting (1) with going to Middle, which is worth $\gamma V^*(\text{Middle})$:

$$V^*(\text{Start}) = \max\{1,\ \gamma\, V^*(\text{Middle})\}.$$

With $\gamma = 0.9$ this gives $V^*(\text{Middle}) = \max\{2, 3.6\} = 3.6$ and $V^*(\text{Start}) = \max\{1, 3.24\} = 3.24$, the best policy's value. Solving a problem backward, from the end, by asking at each decision "what is the best I can do from here, given that I will act well afterward?" is *Bellman's principle of optimality*. The equations above are the simplest instance of the *Bellman optimality equations* of Chapter 3, and computing them backward is *dynamic programming* (Chapter 4). Chapter 3 also proves the general version of Proposition 1.1: in every finite problem of the kind we will study, with discounting and a fully observed state, some deterministic policy is optimal. Randomness in a policy is useful for exploring and for learning, but not, in these problems, for the final behavior. (It *is* useful in games against opponents, Chapter 21, and when the agent cannot see the state, Chapter 19.)

1.9 What is a reward? ●●●

Everything in this book rests on a single scalar reward. Is that enough to express what we want from an agent? The working assumption of the field is often called the *reward hypothesis*: that all of what we mean by goals and purposes can be thought of as maximizing the expected cumulative sum of a scalar reward signal (Sutton and Barto 2018, sec. 3.2). Silver et al. (2021) went further, arguing that maximizing reward in a rich enough environment could by itself give rise to the abilities associated with intelligence, such as perception, language and social skills. The claim is provocative, and it has been contested.

One line of criticism concerns *what a single number can encode*. Many real goals have several components that cannot be traded against each other at a fixed exchange rate, such as performance, safety and fairness; Vamplew et al. (2022) argue that such goals call for vector-valued rewards and multi-objective methods. Constrained formulations, which maximize one quantity subject to bounds on others, are one practical answer (Chapter 23).

A second line asks precisely *which tasks a Markov reward can express*: a reward that depends only on the current state and action, as in this book. Abel et al. (2021) formalize a task as a set of acceptable policies, or as an ordering over policies or over trajectories, and show that some tasks cannot be captured by any Markov reward in a given environment. A simple example appears in Exercise 1.14: in an environment with one state and two actions, no Markov reward makes "alternate between the two actions" strictly better than every other behavior, although a reward that also looks at the previous action can. The fix, enlarging the state, is always available but is not free: it makes the problem larger, and choosing the right enlargement is itself a design problem. Bowling et al. (2023) give axioms on an agent designer's preferences under which goals *can* be expressed as cumulative rewards, which clarifies what the hypothesis actually assumes.

A third line is practical. Rewards are usually written by people, and optimizing a proxy that is slightly wrong can produce behavior that is very wrong, a phenomenon called *reward hacking* or *specification gaming*. With language models, where rewards are often themselves learned from human judgments, this is a central difficulty (Chapter 22). Learning rewards from demonstrations or preferences instead of writing them by hand is the subject of Chapter 18.

> **RESEARCH NOTE**
>
> Open questions: Which goals that people care about can be expressed as Markov rewards over a feasible state, and how large must the state be? How should an agent behave when its reward is known to be an imperfect proxy? When rewards come from learned models of human preferences, how can we detect and prevent overoptimization? Chapters 22–24 return to these questions.

1.10 A short history

Modern reinforcement learning grew from three threads that developed separately for decades and came together in the 1980s (Sutton and Barto 2018, sec. 1.7).

The first thread is learning by trial and error in psychology, from Thorndike's law of effect (Thorndike 1911) to the study of conditioning, and its early computational versions. Minsky's 1961 essay on artificial intelligence discussed reinforcement learning and named the credit assignment problem (Minsky 1961).

The second thread is optimal control and dynamic programming. In the 1950s Richard Bellman introduced the equations that bear his name, the principle of optimality, and Markov decision processes as their setting, and coined the phrase "curse of dimensionality" for the explosion of computation as problems grow (Bellman 1957b). Ronald Howard's policy iteration followed in 1960 (Howard 1960). These methods assumed a known model and did not learn.

The third thread is temporal-difference learning: learning a prediction from a later, better-

informed prediction. Arthur Samuel's checkers program of 1959 already improved its evaluation of positions this way (Samuel 1959). Barto, Sutton and Anderson's actor-critic system learned to balance a pole on a cart in 1983 (Barto et al. 1983), and Sutton formalized temporal-difference methods in 1988 (Sutton 1988). In 1989 Chris Watkins brought the threads together with *Q-learning*, which learns optimal behavior from experience without a model (Watkins 1989; Watkins and Dayan 1992).

The early 1990s brought the first great success: Gerald Tesauro's TD-Gammon learned to play backgammon close to the level of the best human players by playing against itself (Tesauro 1995). Around the same time, a mathematical theory of the methods developed, connecting them to stochastic approximation and dynamic programming (Bertsekas and Tsitsiklis 1996).

The modern era of deep reinforcement learning began when neural networks were combined with Q-learning to play Atari video games from raw pixels: the DQN agent reached a level comparable to a professional human tester on many of 49 games (Mnih et al. 2015). AlphaGo, which combined deep networks, reinforcement learning and tree search (Silver et al. 2016), defeated the world-class player Lee Sedol in 2016; its successors learned from self-play alone (Silver, Schrittwieser, et al. 2017), mastered chess and shogi with the same algorithm (Silver et al. 2018), and finally learned without being told the rules (Schrittwieser et al. 2020). Agents reached grandmaster level at the real-time strategy game StarCraft II (Vinyals et al. 2019) and defeated the world champions of the team game Dota 2 (Berner et al. 2019). Reinforcement learning also left the game board: it has navigated stratospheric balloons (Bellemare et al. 2020), shaped plasma in a fusion reactor (Degrave et al. 2022), outraced champion drivers in a realistic driving simulator (Wurman et al. 2022), and beaten human champions at drone racing (Kaufmann et al. 2023).

The most visible recent use is in language models. Reinforcement learning from human feedback, developed for simple tasks (Christiano et al. 2017) and then for summarization (Stiennon et al. 2020), became a standard step in training conversational assistants (Ouyang et al. 2022). Since 2024, models trained with reinforcement learning on problems whose answers can be checked automatically, such as mathematics and programming, have learned to produce long chains of reasoning (Guo et al. 2025). These developments, and their open problems, are the subject of Chapter 22.

> **HISTORICAL NOTE**
>
> The name "reinforcement" comes from the psychology of animal learning, where a *reinforcer* is a consequence that makes the behavior it follows more likely. Pavlov used the word for conditioned reflexes; Thorndike's law of effect described the same idea for voluntary actions.

1.11 How this book is organized

The book has seven parts (Figure 1.6). Each part builds on the ones before it, but after Part II you can follow your interests.

- Part I, Foundations, defines the problem precisely. Chapter 2 studies the simplest case,

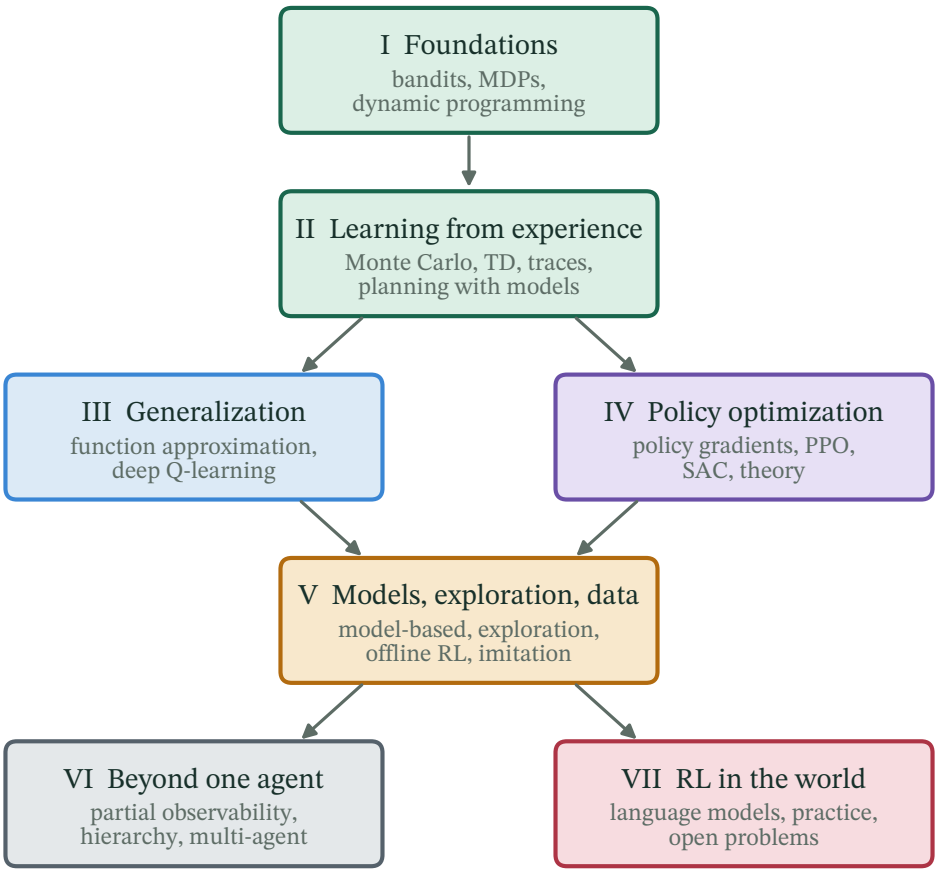


Figure 1.6. The map of the book. Parts I and II build the foundations that everything else uses; after them, generalization (Part III) and policy optimization (Part IV) can be read in either order, and Parts V–VII draw on both.

a single repeated choice (bandits), where exploration can be analyzed exactly. Chapter 3 introduces Markov decision processes, value functions and the Bellman equations, and Chapter 4 shows how to solve them when the model is known.

- Part II, Learning from experience, drops the model: Monte Carlo methods (Chapter 5), temporal-difference learning (Chapter 6), their combination in multi-step methods (Chapter 7), and learning a model to plan with (Chapter 8).
- Part III, Generalization, replaces tables by function approximators, from linear functions (Chapter 9) to deep neural networks (Chapter 10), and explains when and why this can go wrong.
- Part IV, Policy optimization, learns policies directly: policy gradients and actor-critic methods (Chapter 11), trust regions and PPO (Chapter 12), off-policy methods for continuous control such as SAC (Chapter 13), and the theory of why these methods work (Chapter 14).
- Part V, Models, exploration and data, covers model-based deep RL (Chapter 15), exploration (Chapter 16), learning from fixed data sets (Chapter 17), and learning from demonstrations (Chapter 18).
- Part VI, Beyond a single agent and a known state, treats partial observability (Chapter 19), hierarchy and goals (Chapter 20), and multiple agents and games (Chapter 21).
- Part VII, Reinforcement learning in the world, covers RL for language models (Chapter 22), the craft of making RL work and evaluating it honestly (Chapter 23), and the open problems of the field (Chapter 24).

Three reading tracks. Every section is marked with its level: unmarked or ● for the Foundations track, ●● for the Graduate track, ●●● for the Research track. The ● sections alone form a complete introductory course; the ●● sections add the derivations, proofs and full modern algorithms of a graduate course; the ●●● sections lead into current research. The front matter ("How to Use This Book") suggests routes through the book for different readers and courses.

Labs and exercises. Every chapter has labs in the `labs/` folder: short Python programs, built on a small library (`rlcourse`) that implements the book's algorithms in plain NumPy, that you can run, read and modify. Every chapter ends with exercises at all three levels, and the companion volume contains complete solutions.

1.12 Summary

- Reinforcement learning is learning to act from evaluative feedback: an agent interacts with an environment, receives rewards, and adapts its policy to maximize the expected return, the discounted sum of future rewards.
- The agent–environment loop (state, action, reward, next state) is the common structure of problems as different as games, robot control, recommendation and training language models.
- On the Trail, the best behavior forgoes immediate rewards for a larger delayed one, and which behavior is best depends on the discount factor: patience pays only when $\gamma > 0.5$.
- The value of a state under a policy is its expected return. For the Trail's uniform policy, $V^{\pi}(\text{Start}) = 1.76$; the best policy earns 3.24.

- Reinforcement learning is hard because of delayed consequences (credit assignment), the need to explore, the need to generalize to unseen states, and because the agent's behavior determines its own data.
- Agents learn policies, value functions, models, or combinations of them; the main families are value-based, policy-based, actor-critic and model-based methods.
- Values can be estimated by averaging returns, with an error that shrinks like $1/\sqrt{n}$; learning to act requires exploring, or the agent may never discover its best options.
- (●●) The best policy of the Trail is deterministic and can be computed backward from the end, an instance of Bellman's principle of optimality.

The next chapter studies the exploration–exploitation dilemma in its simplest setting, where there is a single state and the only question is which action to choose.

1.13 Further Reading

The classic introduction is Sutton and Barto (2018); its first chapter covers the same ground as this one from a slightly different angle, and its Section 1.7 tells the early history in detail. Szepesvári (2010) is a concise, mathematically careful overview of the core algorithms, and Bertsekas (2019) presents reinforcement learning from the viewpoint of optimal control. The survey by Kaelbling et al. (1996) gives a snapshot of the field before deep learning that is still a good read.

For the landmark results of deep reinforcement learning, the original papers are readable: Mnih et al. (2015) for Atari, Silver et al. (2016) and Silver et al. (2018) for Go, chess and shogi. For the connection between temporal-difference learning and dopamine, see Schultz et al. (1997).

The debate about rewards (Section 1.9) is best followed in the original papers: Silver et al. (2021), the reply by Vamplew et al. (2022), the analysis of what Markov rewards can express by Abel et al. (2021), and the axiomatic treatment by Bowling et al. (2023).

1.14 Exercises

Exercise 1.1 ● Agents, environments, rewards

For each of the following, describe a reasonable choice of agent, environment, state, actions and reward, and say whether the task is episodic or continuing: (a) a thermostat that controls a home's heating; (b) a program that plays chess; (c) a system that chooses which advertisement to show on a web page; (d) a robot vacuum cleaner.

Exercise 1.2 ● Returns with a smaller discount

Compute the discounted returns of the Trail's three paths when $\gamma = 0.8$. Which path is best? What if $\gamma = 0.4$?

Exercise 1.3 ● When does patience pay?

Find, as functions of γ, the conditions under which (a) going on and exiting at Middle beats exiting at Start; (b) going all the way beats going on and exiting at Middle; (c) going all the way beats exiting at Start. Explain why all three thresholds are the same.

Exercise 1.4 ● The value of a policy

Compute the expected return of the Trail policy with $p_0 = 0.8$ and $p_1 = 0.5$ when $\gamma = 0.9$, first by listing paths and then with the formula (1.2).

Exercise 1.5 ● Learning what you never try

An agent follows a policy with $p_0 = 0$. Explain why, however many episodes it plays, it cannot learn anything about the value of Goal. Propose two different ways of changing its behavior so that it can, and say what each costs.

Exercise 1.6 ● Which kind of learning?

Classify each problem as supervised learning, unsupervised learning, reinforcement learning or planning, and justify your answer: (a) detecting spam from a set of labeled emails; (b) grouping customers by purchasing behavior; (c) choosing the dose of a drug for a patient over several visits, based on how the patient responds; (d) finding the shortest route on a known road map; (e) learning to play a new video game from the score alone.

Exercise 1.7 ● Designing a reward

Propose a reward function for a robot vacuum cleaner whose job is to keep a house clean. Then describe a behavior that would earn a high reward under your function but that the owner would not want. Modify the reward to rule it out, and look for a new loophole.

Exercise 1.8 ●● Returns are bounded

Suppose every reward satisfies $|r_t| \le r_{\max}$. (a) Show that $|G_t| \le r_{\max}/(1-\gamma)$ when $\gamma < 1$. (b) The Trail is well defined even with $\gamma = 1$; compute the three returns in that case and explain why no discounting is needed. (c) Give an example of a continuing task in which $\gamma = 1$ makes the return undefined.

Exercise 1.9 ●● Every policy ties

Show that when $\gamma = 0.5$, every policy of the Trail, deterministic or stochastic, has expected return exactly 1. What does this say about the policy that a learning agent will find at this discount factor?

Exercise 1.10 ●● How many episodes?

For the uniform policy with $\gamma = 0.9$: (a) compute the variance of the return G_0; (b) using the normal approximation, how many episodes are needed for the average return to be within ± 0.05 of 1.76 with probability about 95%? (c) Compare with the band in Figure 1.4.

Exercise 1.11 ●● Episode length

Find the expected number of steps in an episode of the Trail as a function of p_0 and p_1. What is it for the uniform policy, and which policy has the longest expected episodes?

Exercise 1.12 ●● Deciding backward

A *decision tree* is a problem in which every state is visited at most once per episode and every episode ends after at most H steps (the Trail is one, with $H = 3$). Sketch an argument that some deterministic policy is optimal for every decision tree, by deciding the last choices first, as in Section 1.8. Where does the argument use the assumption that states are not revisited?

Exercise 1.13 ● Learning both decisions

Extend Task 5 of Lab 1.1 so that the agent also learns what to do at Middle, keeping one running average per state and action of the returns that followed. With $\varepsilon = 0.1$, how many episodes does the agent need before it reliably goes all the way? What happens with $\varepsilon = 0$?

Exercise 1.14 ●●● A goal that no Markov reward can express

Consider an environment with a single state and two actions, L and R; the episode never ends and $0 < \gamma < 1$. The designer wants the agent to alternate, L, R, L, R, ... (a) Show that for every reward function r(L), r(R) that depends only on the action, some policy that does not alternate earns at least as much expected return as the alternating policy. (b) Enlarge the state to record the previous action and give a reward under which alternating is the unique optimal behavior. (c) Relate your answer to the discussion of Abel et al. (2021) in Section 1.9.

Exercise 1.15 ●●● Is reward enough?

Read Silver et al. (2021) and the reply by Vamplew et al. (2022). Summarize each side's main argument in a paragraph, identify one point on which they actually agree, and argue for your own position, with an example from an application you know.

CHAPTER 2

Bandits: Learning to Choose

WHAT YOU WILL LEARN

- Formulate a multi-armed bandit problem and measure a learning algorithm by its regret.
- Estimate action values incrementally, with sample averages or a constant step size, and recognize the update "old estimate plus step size times error" that recurs throughout the book.
- Explain why greedy action selection fails, and how ε-greedy selection, optimistic initial values, upper confidence bounds and Thompson sampling explore.
- Derive the gradient-bandit update as stochastic gradient ascent on the expected reward, and explain why a baseline leaves it unbiased.
- Prove a logarithmic regret bound for UCB1, and state the lower bounds that limit every algorithm (●●).
- Extend upper confidence bounds to contextual bandits with linear rewards (●●).
- Survey asymptotically optimal algorithms, adversarial bandits, pure exploration and Bayesian optimality (●●●).

A doctor has two treatments for a disease, and each new patient must receive one of them. She wants to cure as many patients as she can, but nobody knows which treatment is better. Each patient who gets the treatment that currently looks better is a chance to help that patient; each patient who gets the other one is a chance to learn something that may help the patients who come later. If she always prescribes the treatment that looks better, an unlucky start can lead her to prescribe the worse one forever. If she keeps alternating, she learns a lot and cures fewer people than she could. This tension between *exploiting* what you know and *exploring* to know more is the exploration–exploitation dilemma that Chapter 1 named as one of the core difficulties of reinforcement learning.

The same structure appears in many places: a website that tests several headlines on its readers, an advertising system that chooses which ad to show, a recommender that picks a song, a lab that allocates experiments among candidate materials. It even appeared at the end of Chapter 1, where an agent on the Trail had to choose at Start between exiting for a sure +1 and going on to a reward it had never seen. In all of these the decision maker faces the same choice over and over, the outcome of each choice is random, and the only way to learn what an option is worth is to try it.

This chapter studies the simplest problem with this structure, the *multi-armed bandit*. The name comes from slot machines, once called "one-armed bandits": picture a row of machines with different, unknown payout rates, and a gambler who must decide which arm to pull next. A

bandit has no states and no delayed consequences, so it isolates exploration from everything else that makes reinforcement learning hard. That is why it comes first. Many ideas of the whole book make their first appearance here: estimates updated by "old estimate plus step size times error", ε-greedy exploration, optimism, posterior sampling, and softmax policies trained by gradient ascent.

Section 2.1 defines the problem and the measure of performance, regret. Section 2.2 shows how to estimate the value of each arm. Section 2.3 to 2.6 present four ways to choose arms: ε-greedy selection, upper confidence bounds, Thompson sampling and gradient bandits. Section 2.7 compares them and gives practical advice. These ● sections form a complete introduction. The ●● sections prove that UCB1's regret grows only logarithmically (Section 2.8), show that no algorithm can do much better (Section 2.9), and add context to the choices (Section 2.10). Section 2.11 (●●●) surveys the research frontier. On a first reading, read the ● sections and run the lab.

2.1 The multi-armed bandit problem

Bandit problems were first studied for medicine. In 1933 William Thompson asked how to allocate patients between two treatments so that the better one is found while as few patients as possible receive the worse one (Thompson 1933). Herbert Robbins posed the general question of sequential experimentation in 1952 (Robbins 1952). Today the most common applications are online: choosing which version of a web page, an advertisement or a recommendation to show, where millions of rounds are played every day and each click is a reward.

All of these share the same protocol. There are K options, called *arms*. In each round the agent chooses one arm and receives a random reward from it. It sees only the reward of the arm it chose, not what the other arms would have paid. The rewards of each arm come from a fixed distribution that the agent does not know.

Definition 2.1 Stochastic multi-armed bandit

A *K-armed stochastic bandit* has arms $a \in \{1, \dots, K\}$, each with an unknown reward distribution with mean μ_a. In each round $t = 1, 2, \dots, n$ the agent chooses an arm a_t, using only the arms and rewards of earlier rounds, and receives a reward r_t drawn from the distribution of arm a_t, independently of everything that happened before.

We number rounds from $t = 1$ in this chapter, as the bandit literature does. Three quantities describe a bandit. The *best mean* is $\mu^* = \max_a \mu_a$, and an arm that attains it is an *optimal arm*, written a^*. The *gap* of arm a is $\Delta_a = \mu^* - \mu_a \geq 0$: how much is lost, on average, each time arm a is pulled instead of an optimal arm. Finally, $N_a(t)$ counts how many times arm a was chosen in the first t rounds.

In the language of Chapter 1, a bandit is a reinforcement-learning problem with a single state. Its arms are the actions, and the mean μ_a is the action value $Q(a)$ of that single state; we use the bandit literature's symbol μ_a in this chapter. Because there is only one state, an action has no effect on future situations, only on the reward of the current round and on what the agent learns.

2.1.1 Regret

The agent wants to maximize its expected total reward $\mathbb{E}[\sum_{t=1}^{n} r_t]$. To compare algorithms across problems it is more useful to measure what they lose by not knowing the best arm. An oracle that knew the means would pull an optimal arm in every round and collect $n\mu^*$ in expectation. The shortfall is the regret.

Definition 2.2 Regret

The *regret* of an algorithm after n rounds is

$$\mathrm{Reg}_n = n\mu^* - \mathbb{E}\Big[\sum_{t=1}^{n} r_t\Big],$$

where the expectation is over the random rewards and over any randomness in the algorithm's choices.

In words: regret is the expected reward lost, compared with always playing the best arm. Small regret means the agent found the best arm quickly and then mostly played it. The regret is never negative, because no algorithm can beat the oracle on average.

Regret has a simple and very useful form: it adds up the gaps of the arms the agent pulled.

Proposition 2.3 Regret decomposition

For every algorithm and every n,

$$\mathrm{Reg}_n = \sum_{a=1}^{K} \Delta_a \, \mathbb{E}[N_a(n)]. \tag{2.1}$$

Proof. Given the arm a_t chosen in round t, the reward is a fresh draw from that arm, so $\mathbb{E}[r_t \mid a_t] = \mu_{a_t}$, and by the tower rule $\mathbb{E}[r_t] = \mathbb{E}[\mu_{a_t}]$. Summing over rounds and grouping the rounds by arm, $\mathbb{E}[\sum_t r_t] = \mathbb{E}[\sum_a \mu_a N_a(n)] = \sum_a \mu_a \mathbb{E}[N_a(n)]$. Since $\sum_a N_a(n) = n$, also $n\mu^* = \sum_a \mu^* \mathbb{E}[N_a(n)]$. Subtracting gives $\sum_a (\mu^* - \mu_a)\mathbb{E}[N_a(n)]$. ∎

The decomposition says that each pull of arm a costs Δ_a, so a good algorithm must pull each suboptimal arm rarely, and the worse the arm, the more rarely. It also shows the two regimes that matter. An algorithm that keeps pulling some suboptimal arm in a fixed fraction of the rounds has regret that grows *linearly* in n: it never stops paying. An algorithm whose regret grows *sublinearly*, for instance like $\ln n$, pulls suboptimal arms in a vanishing fraction of the rounds and eventually plays almost as well as the oracle. The rest of the chapter is largely about achieving the second regime.

Example 2.1 The slippery Trail as a bandit

Take the Trail of Chapter 1 and make it slippery, as Chapter 3 will do: every `go` fails with probability 0.2 and leaves the hiker where she is. Let $\gamma = 0.9$. Suppose the hiker treats each episode

as one round and chooses, at the start, one of three plans: exit at Start; go, then exit at Middle; or go all the way. Each plan is an arm, and its reward is the discounted return of the episode, which is random because of the slips. Exiting at once pays 1. The other two plans pay $2\gamma^{1+S}$ and $4\gamma^{2+S}$, where S is the total number of slips; each slip delays the reward by one step. Averaging over the slips (the lab computes these values exactly) gives the arm means $\mu = (1, 1.756, 3.084)$, so the gaps are $\Delta = (2.084, 1.328, 0)$.

A hiker who rotates through the three plans for 300 episodes pulls each arm 100 times, and by (2.1) her regret is $100 \times 2.084 + 100 \times 1.328 = 341.2$, more than a third of the 925 that the best plan would earn in expectation. A hiker who tries each plan a few times and then commits to the best one has much smaller regret, but she risks committing to the wrong plan after an unlucky start. Finding the right balance is the subject of this chapter.

Treating whole plans as arms ignores the structure of the Trail: the second and third plans share their first step, and what the hiker learns about Middle on one plan is useful for the other. Chapter 3 introduces the model that captures this structure, and Chapter 16 returns to exploration with it.

♦

CHECKPOINT

1. A three-armed bandit has means 0.2, 0.5 and 0.7. In 1000 rounds an algorithm pulls the arms 100, 200 and 700 times in expectation. What is its regret?
2. Why must every algorithm that has small regret on *every* bandit pull every arm at least occasionally?

Answers.

1. The gaps are 0.5, 0.2 and 0, so the regret is $100 \times 0.5 + 200 \times 0.2 = 90$.
2. An algorithm that never pulls some arm cannot tell a bandit in which that arm is the best from one in which it is not; on the first kind its regret grows linearly.

2.2 Estimating action values

Before an agent can choose well, it needs an estimate of what each arm is worth. The natural estimate of a mean is an average. If arm a has been pulled N_a times, its *sample average* is

$$\hat{\mu}_a = \frac{\text{sum of the rewards received from arm } a}{N_a}.$$

By the law of large numbers, $\hat{\mu}_a$ converges to μ_a as N_a grows. An arm that has never been pulled needs some initial guess; we come back to that choice in Section 2.3.

2.2.1 The incremental update

Storing every reward to recompute the average would waste memory and time. Fortunately the average can be updated one reward at a time. Let $\hat{\mu}_k$ be the average of an arm's first k rewards

$r_1, \dots, r_k$, and let r_{k+1} be its next reward. Then

$$\hat{\mu}_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = \frac{1}{k+1}\big(r_{k+1} + k\,\hat{\mu}_k\big) = \hat{\mu}_k + \frac{1}{k+1}\big(r_{k+1} - \hat{\mu}_k\big).$$

This has a form that will appear again and again in this book:

$$\hat{\mu} \leftarrow \hat{\mu} + \eta\,\big(r - \hat{\mu}\big), \tag{2.2}$$

with *step size* $\eta = 1/(k+1)$ for the sample average. In words: the new estimate is the old estimate plus a step toward the new observation. The difference $r - \hat{\mu}$ is the *error* of the old estimate on the new observation, and the step size says how far to move.

The general pattern, *new estimate = old estimate + step size × (target − old estimate)*, is the backbone of reinforcement learning. In Monte Carlo methods (Chapter 5) the target is an observed return. In temporal-difference learning (Chapter 6) the target is itself built from another estimate. In deep reinforcement learning (Chapter 10) the same correction becomes the gradient of a squared loss. Each time, the estimate moves toward a target in proportion to its error.

Example 2.2 Incremental averaging by hand

An arm returns the rewards $1, 0, 1, 1$. With step size $1/k$ the estimates are

$$1, \qquad 1 + \tfrac{1}{2}(0-1) = 0.5, \qquad 0.5 + \tfrac{1}{3}(1-0.5) = 0.667, \qquad 0.667 + \tfrac{1}{4}(1-0.667) = 0.75,$$

exactly the running averages. With a constant step size $\eta = 0.5$ and an initial estimate of 0, the same rewards give 0.5, 0.25, 0.625 and 0.8125. The last estimate is a weighted average of the initial value and the four rewards with weights 0.0625, 0.0625, 0.125, 0.25 and 0.5: the latest reward counts the most.

♦

2.2.2 Constant step sizes and changing worlds

The sample average weighs all rewards equally. That is right when the arm's distribution never changes, but many real problems drift: the popularity of a headline fades, a machine wears out, a market moves. For such *non-stationary* problems we want recent rewards to count more than old ones, and a constant step size $\eta \in (0, 1]$ does exactly that. Unrolling the update (2.2) for n rewards, starting from an initial estimate $\hat{\mu}_0$, gives

$$\hat{\mu}_n = (1-\eta)^n\,\hat{\mu}_0 + \sum_{i=1}^{n} \eta(1-\eta)^{n-i}\,r_i. \tag{2.3}$$

The weights $\eta(1-\eta)^{n-i}$ decay geometrically into the past, which is why this is called an *exponential recency-weighted average*. The weights, including the weight $(1-\eta)^n$ of the initial estimate, add up to one (Exercise 2.3). With $\eta = 0.1$, a reward received ten pulls ago carries weight $0.1 \times 0.9^{10} \approx 0.035$, and the initial estimate has lost almost two thirds of its influence after ten pulls ($0.9^{10} \approx 0.35$).

The price of forgetting is that the estimate never settles down. With a constant step size and a stationary arm whose rewards have variance σ^2, the estimate keeps fluctuating around μ_a with variance $\eta\sigma^2/(2-\eta)$ in the long run. With step sizes $1/k$ it converges. More generally, an estimate updated by (2.2) with step sizes η_k converges to the mean if $\sum_k \eta_k = \infty$ and $\sum_k \eta_k^2 < \infty$: the steps must be large enough to overcome any initial error, yet shrink fast enough to average out the noise. These are the *Robbins–Monro conditions* of stochastic approximation, stated precisely in Theorem B.1 of Appendix B. The step size $1/k$ satisfies both; a constant step size violates the second on purpose, so that it can keep tracking a moving target.

> **PITFALL**
>
> A constant step size is not a "better average". On a stationary problem it throws information away: its estimates stay noisy no matter how much data arrives. Use sample averages (step size $1/N$) when you believe the arms do not change, and a constant step size when you believe they do. Lab Task 6 measures the difference on a drifting bandit.

2.3 Greedy and ε-greedy action selection

With estimates in hand, the obvious rule is to pull the arm whose estimate is highest. This is the *greedy* rule:

$$a_t = \arg\max_a \hat{\mu}_a,$$

with ties broken at random. Greedy selection exploits perfectly and explores not at all. Its weakness is that an estimate can only improve if its arm is pulled, and a greedy agent stops pulling any arm that looks bad, even if it looks bad only because of bad luck.

Example 2.3 How greed locks in the worse treatment

Two treatments cure a patient with probabilities $\mu_A = 0.4$ and $\mu_B = 0.6$, so each round's reward is 1 (cured) or 0. A greedy agent starts with both estimates at 0. As long as both estimates are 0 it picks an arm at random. The first time an arm produces a cure, its estimate becomes positive, and it stays positive forever, because an average that includes a 1 cannot fall back to 0. The other arm, which so far has given only zeros or has never been tried, keeps the estimate 0 and is never chosen again.

So the agent commits forever to whichever arm produces the first cure. In each round before that, arm A is chosen with probability $\frac{1}{2}$ and cures with probability 0.4, while arm B is chosen with probability $\frac{1}{2}$ and cures with probability 0.6. The first cure comes from A with probability

$$\frac{\frac{1}{2}\times 0.4}{\frac{1}{2}\times 0.4 + \frac{1}{2}\times 0.6} = 0.4.$$

In 40% of its runs the greedy agent prescribes the worse treatment to every patient, forever. Its regret grows linearly, about $0.4 \times 0.2 = 0.08$ per round. The lab's tests confirm the 40% by simulation.

The failure is not about these numbers. Whenever the rewards are noisy, there is a positive probability that the best arm looks worse than another arm after its first few pulls, and a greedy agent never collects the evidence that would correct the mistake.

2.3.1 ε-greedy selection

The simplest fix is to explore a little all the time. An *ε-greedy* agent picks a uniformly random arm with probability ε and the greedy arm otherwise.

Algorithm 2.1 ε-greedy bandit

Input: number of arms K, exploration rate $\varepsilon \in [0, 1]$, step size rule ($\eta = 1/N_a$ or a constant), initial estimate $\hat{\mu}_0$.

1. Set $\hat{\mu}_a \leftarrow \hat{\mu}_0$ and $N_a \leftarrow 0$ for every arm a.
2. For $t = 1, 2, \dots, n$:
   1. With probability ε, let a_t be a uniformly random arm; otherwise let $a_t = \arg\max_a \hat{\mu}_a$ (ties broken at random).
   2. Pull a_t and observe the reward r_t.
   3. $N_{a_t} \leftarrow N_{a_t} + 1$.
   4. $\hat{\mu}_{a_t} \leftarrow \hat{\mu}_{a_t} + \eta\,(r_t - \hat{\mu}_{a_t})$, with $\eta = 1/N_{a_t}$ for sample averages.

Output: the estimates $\hat{\mu}_a$ and the sequence of rewards.

Because every arm is pulled with probability at least ε/K in every round, every arm is pulled infinitely often, every sample average converges to its mean, and eventually the greedy arm is an optimal arm. From then on the agent picks the optimal arm with probability $1 - \varepsilon + \varepsilon/K$: the greedy pick, plus the chance that the random pick happens to land on it. With $\varepsilon = 0.1$ and $K = 10$, that is 0.91.

The same computation exposes the weakness of a constant ε: the agent never stops exploring. In the long run it pulls each suboptimal arm in a fraction ε/K of the rounds, so its regret grows linearly, at the rate $\frac{\varepsilon}{K}\sum_a \Delta_a$ per round (Exercise 2.5). This is much better than greedy's linear regret, whose rate can be as large as the largest gap, but it is still linear.

2.3.2 The 10-armed testbed

How do these rules compare in practice? A standard experiment, from Sutton and Barto (2018, sec. 2.3), is the *10-armed testbed*. Each problem is a bandit with $K = 10$ arms whose means μ_a are drawn from a standard normal distribution; each pull returns the arm's mean plus standard normal noise. Every algorithm is run for 1000 rounds on each of 500 such problems, and the results are averaged. The best possible average reward is the expected largest of ten standard normal numbers, about 1.54.

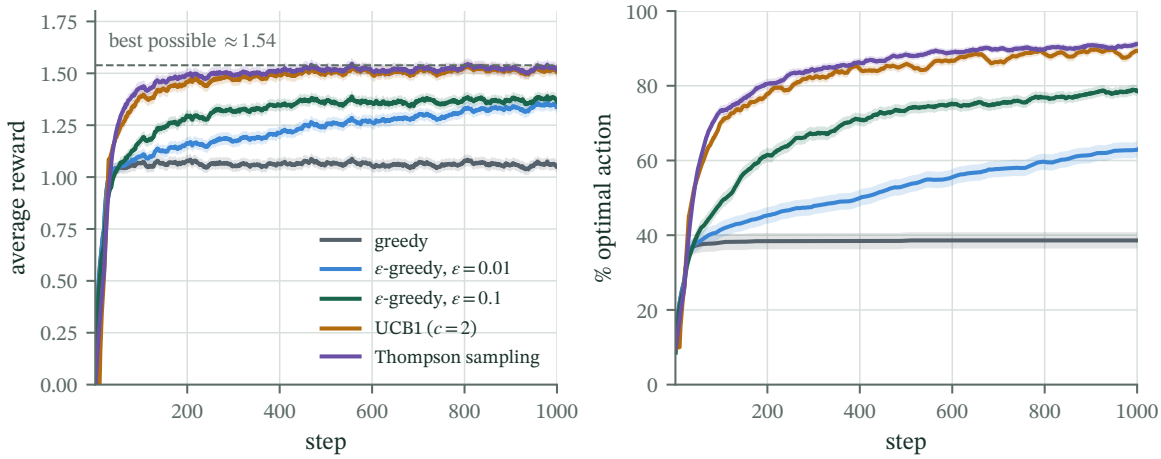


Figure 2.1. The 10-armed testbed: average reward (left) and percentage of rounds in which the optimal arm was chosen (right), averaged over 500 random problems and smoothed over 20 rounds; the bands show ±1 standard error. Greedy settles quickly and badly. A little exploration ($\varepsilon = 0.01$ or 0.1) keeps improving. UCB1 and Thompson sampling, introduced in the next two sections, explore in a directed way and do best.

Figure 2.1 shows the result. Greedy selection improves fastest at the very start, then stalls: over the last 100 rounds it earns an average reward of 1.06 and plays the optimal arm in only 39% of the problems. With $\varepsilon = 0.1$ the agent finds the best arm in more problems (78% optimal over the last 100 rounds, average reward 1.37), but it can never exceed 91% optimal play because of its random pulls. With $\varepsilon = 0.01$ it learns more slowly (63% at the end) but keeps improving, and in a long enough run it would overtake $\varepsilon = 0.1$. Which ε is best depends on the horizon: Lab Task 2 finds that over 1000 rounds the average reward peaks between $\varepsilon = 0.03$ and 0.1.

2.3.3 Decaying exploration

A natural compromise is to explore a lot at first and less later, with an exploration rate ε_t that decreases with t. If ε_t decreases slowly enough that every arm is still pulled infinitely often, and tends to zero so that the agent eventually acts greedily, the agent is *greedy in the limit with infinite exploration* (GLIE), a property that Chapter 5 uses for control. With ε_t proportional to $1/t$, the expected number of random pulls up to round n grows only like $\ln n$, and Auer, Cesa-Bianchi, and Fischer (2002) showed that the schedule $\varepsilon_t = \min\{1, cK/(d^2 t)\}$, with a constant $c > 5$ and d a lower bound on the smallest positive gap, makes the regret logarithmic. The catch is that the schedule must know d: decay too fast and the agent may stop exploring before it has found the best arm; decay too slowly and it explores for too long. The methods of the next sections adapt their exploration to the data instead.

2.3.4 Optimistic initial values

There is another way to make even a greedy agent explore: start with estimates that are too high. Suppose every estimate starts at $\hat{\mu}_0 = 5$ on the testbed, where no arm's mean is likely to exceed 3, and the agent uses a constant step size $\eta = 0.1$. Whichever arm it pulls first returns something

far below 5, its estimate drops, and the agent turns to an arm that still has its optimistic estimate. The agent is "disappointed" by every arm in turn and so tries all of them, then keeps trying the ones whose estimates are still inflated. In Lab Task 3 this optimistic greedy agent plays the optimal arm in 85% of the last 100 rounds, against 76% for a realistic ε-greedy agent with $\varepsilon = 0.1$ and the same step size.

Optimistic initial values are a simple and useful trick, but a limited one. The extra exploration happens only at the beginning, so it does not help if the problem changes later. It requires knowing roughly how large rewards can be. And with sample averages it lasts only one pull per arm, because the first update with step size 1/1 replaces the initial value entirely. The idea behind it is more important than the trick: *treat what you do not know as if it were good*, so that the agent is drawn toward uncertain options. The next section turns this idea into a principled algorithm.

TRY IT INTERACTIVELY

The interactive course (`Apps_claude/RL Course/index.html`, Module 5, lesson "Exploration: trying what looks worse") shows a greedy learner trapping itself on the Trail, and lets you switch on ε-greedy exploration or optimistic starting values and test a thousand learners at once.

CHECKPOINT

1. With $K = 4$ arms and $\varepsilon = 0.2$, what is the probability that an ε-greedy agent picks the greedy arm? Any particular other arm?
2. In Example 2.3, what is the probability of locking in the worse arm if $\mu_A = 0.9$ and $\mu_B = 0.95$?
3. Why does a constant $\varepsilon > 0$ give linear regret even after the estimates are accurate?

Answers.

1. $1 - 0.2 + 0.2/4 = 0.85$ for the greedy arm and $0.2/4 = 0.05$ for each other arm.
2. $0.9/(0.9 + 0.95) \approx 0.49$: when both arms are good, the first success says almost nothing, and greed is nearly a coin flip.
3. Because a fraction ε of the rounds are still random, and each random pull of a suboptimal arm costs its gap, round after round.

2.4 Upper confidence bounds

The ε-greedy rule explores blindly. When it explores, it picks any arm with the same probability: an arm that has been pulled a thousand times and is known to be poor gets the same attention as an arm that has been pulled twice. A smarter agent would explore the arms that *might* be the best, and stop exploring arms that are known to be bad.

The guiding principle is called *optimism in the face of uncertainty*: for each arm, consider how good it could plausibly be, given the data so far, and act as if each arm were that good. If the optimistic guess for the chosen arm is right, the agent is exploiting a good arm. If it is wrong, the arm's estimate drops and its uncertainty shrinks, so the agent learns something and soon moves on. Either way the choice pays off. This self-correcting property is what greed lacks.

2.4.1 How uncertain is an estimate?

To be optimistic in a principled way we need to know how far a sample average can be from the true mean. Suppose an arm's rewards lie between 0 and 1, and its average after N pulls is $\hat{\mu}$. *Hoeffding's inequality* (Theorem B.9 in Appendix B) says that the probability that the average underestimates the mean by more than c is small and shrinks exponentially in Nc^2:

$$\Pr\big(\hat{\mu} \le \mu - c\big) \le e^{-2Nc^2}.$$

Turning this around: if we want the probability of an underestimate by more than c to be at most δ, we can take

$$c = \sqrt{\frac{\ln(1/\delta)}{2N}}.$$

So with probability at least $1-\delta$ the true mean lies below the *upper confidence bound* $\hat{\mu} + c$. The radius c shrinks like $1/\sqrt{N}$: four times as many pulls halve the uncertainty. With $N = 100$ pulls and $\delta = 0.05$, for example, $c = \sqrt{\ln 20/200} \approx 0.12$.

Which δ should the agent use? If it wants to be wrong rarely over a long run, it should become more demanding as time goes on. UCB1 takes $\delta = t^{-4}$ in round t, which turns the radius into

$$c = \sqrt{\frac{2\ln t}{N}}.$$

2.4.2 UCB1

Algorithm 2.2 UCB1

Input: number of arms K; rewards in $[0, 1]$.

1. Pull each arm once (rounds $1, \dots, K$), and set $N_a \leftarrow 1$ and $\hat{\mu}_a \leftarrow$ the reward received from arm a.
2. For $t = K+1, K+2, \dots, n$:
   1. For every arm compute the index $U_a = \hat{\mu}_a + \sqrt{2\ln(t-1)/N_a}$, where $t-1$ is the number of rounds played so far.
   2. Pull $a_t = \arg\max_a U_a$ (ties broken at random) and observe r_t.
   3. $N_{a_t} \leftarrow N_{a_t} + 1$ and $\hat{\mu}_{a_t} \leftarrow \hat{\mu}_{a_t} + (r_t - \hat{\mu}_{a_t})/N_{a_t}$.

Output: the sequence of rewards and the final estimates.

UCB1, from Auer, Cesa-Bianchi, and Fischer (2002), is deterministic: it explores only through the bonus $\sqrt{2\ln(t-1)/N_a}$. The bonus does two jobs. It decreases each time arm a is pulled, so an arm that has been tried often must earn its place through a high average. And it increases slowly, like $\sqrt{\ln t}$, while the arm is not pulled, so no arm is ever abandoned for good: an arm that was unlucky early will eventually be retried, but less and less often. The index uses rounds

played so far; implementations, including `rlcourse.bandits.UCB1`, keep a counter t of rounds played and use $\ln t$.

Example 2.4 UCB1 indices by hand

After 20 rounds, a three-armed bandit has the counts and averages below. UCB1 adds the bonus $\sqrt{2\ln 20/N_a}$, with $2\ln 20 = 5.99$:

Table 2.1. UCB1 indices after 20 rounds.

Arm	N_a	$\hat{\mu}_a$	Bonus	Index U_a
1	10	0.60	0.774	1.374
2	6	0.50	0.999	1.499
3	4	0.30	1.224	1.524

UCB1 pulls arm 3, the arm with the *lowest* average, because it has been tried only four times and could still be the best. A greedy agent would pull arm 1. The indices exceed 1, the largest possible reward; that is fine, because the indices are only compared with each other. If arm 3 returns a 0, its average drops to 0.24 and its bonus to 1.10, and in round 22 arm 2 has the highest index.

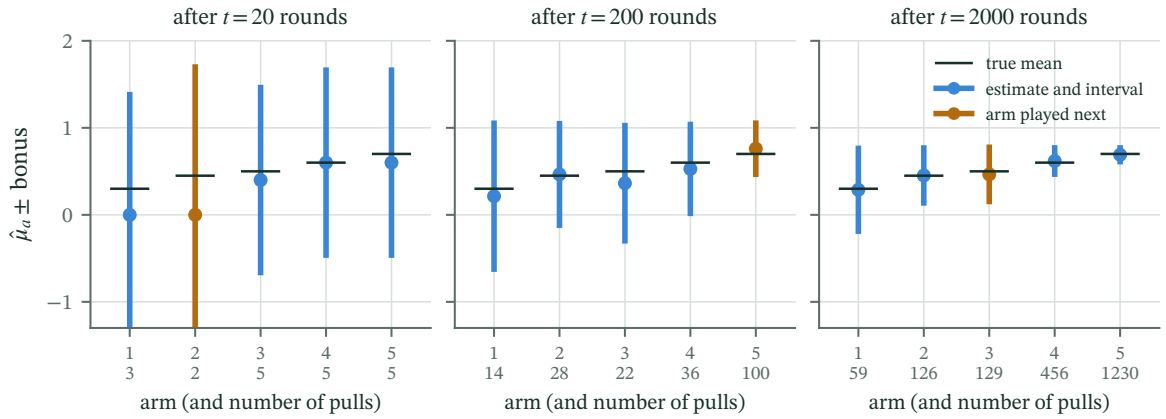


Figure 2.2. UCB1's confidence intervals $\hat{\mu}_a \pm \sqrt{2\ln t/N_a}$ in one run on a five-armed Bernoulli bandit with means 0.3, 0.45, 0.5, 0.6, 0.7 (black ticks). The arm with the highest upper end (amber) is played next. Early on (left) all intervals are wide and UCB1 explores widely. Later, the intervals of the poor arms stay fairly wide because they are rarely played, but their upper ends fall below that of the best arm, which gets most of the pulls (counts under the axis).

Figure 2.2 shows UCB1 at work. Notice that UCB1 does not try to make all intervals small. It only needs the upper end of every poor arm's interval to fall below the best arm's estimate, and arms with large gaps reach that point after fewer pulls. In the right panel, after 2000 rounds, the worst arm (mean 0.3) has been pulled 59 times and the best (mean 0.7) 1230 times.

On the testbed (Figure 2.1), UCB1 plays the optimal arm in 89% of the last 100 rounds and earns 1.51 per round, close to the best possible 1.54, although the testbed's Gaussian rewards are not in $[0, 1]$ as the algorithm assumes. Figure 2.3 compares the regret of ε-greedy, UCB1 and Thompson sampling (the subject of the next section) on a two-armed bandit with the cure rates 0.4 and 0.6 of Example 2.3. With a logarithmic horizontal axis, regret that grows like $\ln n$ becomes a straight line, while linear regret curves sharply upward. The regret of ε-greedy keeps growing at the rate $0.1 \times \frac{1}{2} \times 0.2 = 0.01$ per round, while that of UCB1 bends toward logarithmic growth.

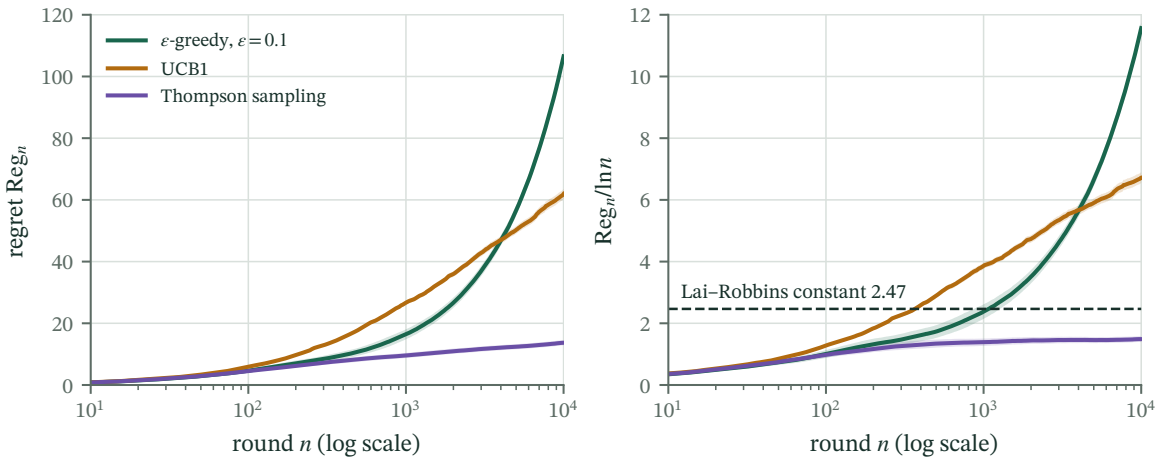


Figure 2.3. Regret on a two-armed Bernoulli bandit with means 0.4 and 0.6, averaged over 100 runs (bands: ±1 standard error). Left: the regret Reg_n against n on a logarithmic axis. The regret of ε-greedy grows linearly and overtakes UCB1's after a few thousand rounds; Thompson sampling has the smallest regret throughout. Right: $\mathrm{Reg}_n / \ln n$, which for the best possible algorithms tends to the Lai–Robbins constant of Section 2.9 (dashed); the convergence is slow.

UCB1 comes with a guarantee that ε-greedy lacks. For rewards in $[0, 1]$, its regret after n rounds is at most

$$\sum_{a:\,\Delta_a > 0} \left(\frac{8 \ln n}{\Delta_a} + \left(1 + \frac{\pi^2}{3}\right) \Delta_a \right),$$

whatever the reward distributions are (Auer, Cesa-Bianchi, and Fischer 2002). Theorem 2.6 states this precisely and Section 2.8 proves it. The bound grows logarithmically in n, and each suboptimal arm contributes in proportion to $1/\Delta_a$: arms that are much worse are discarded quickly, arms that are almost as good as the best cost little per pull. For the two-armed example the bound is $40 \ln n + 0.86 \approx 369$ at $n = 10^4$; the measured regret is 62. The bound is valid but loose, which is typical of worst-case guarantees.

PITFALL

UCB1's bonus $\sqrt{2 \ln t / N}$ assumes rewards in $[0, 1]$. If rewards lie in $[0, R]$, the bonus must be multiplied by R; with Gaussian noise of standard deviation σ the analysis needs $\sqrt{8\sigma^2 \ln t / N}$. A bonus that is too small for the reward scale makes UCB nearly greedy, with the same lock-in failures; one that is too large wastes pulls on poor arms. In practice the constant in front of the bonus is tuned, which is what the parameter c of `rlcourse.bandits.UCB1` is for.

2.5 Thompson sampling

Upper confidence bounds express uncertainty as an interval. A Bayesian agent expresses it as a probability distribution: for each arm, a *posterior* distribution over the arm's unknown mean, given the rewards seen so far. A sharply peaked posterior means the agent is confident; a wide one means it is not. *Thompson sampling* turns this into a decision rule of striking simplicity. In each round, draw one plausible value of every arm's mean from its posterior, and pull the arm whose draw is highest.

Why should this work? An arm is pulled if its draw beats all the others. That happens often for an arm that is probably the best (its posterior sits to the right of the others), sometimes for an arm whose posterior is wide enough that it *might* be the best, and almost never for an arm that is confidently worse. In fact the probability of pulling each arm equals the posterior probability that it is the best arm. Thompson sampling is therefore also called *probability matching*: it explores an arm exactly as much as the agent believes the arm might be optimal.

2.5.1 The Beta–Bernoulli case

The cleanest case is a bandit whose rewards are 0 or 1 (a click or no click, a cure or no cure), so that arm a is described by its success probability $\theta_a = \mu_a$. Here θ_a denotes an unknown parameter of the environment, not of a policy. A convenient prior for a probability is the *Beta distribution* $\mathrm{Beta}(\alpha, \beta)$, whose density is proportional to $\theta^{\alpha-1}(1-\theta)^{\beta-1}$ on $[0,1]$ and whose mean is $\alpha/(\alpha+\beta)$. The uniform distribution is $\mathrm{Beta}(1,1)$.

The Beta prior is *conjugate* to Bernoulli rewards: the posterior is again a Beta distribution. If arm a has produced S_a successes and F_a failures, its likelihood is $\theta^{S_a}(1-\theta)^{F_a}$, and multiplying by a $\mathrm{Beta}(1,1)$ prior gives a density proportional to $\theta^{S_a}(1-\theta)^{F_a}$, which is $\mathrm{Beta}(S_a+1, F_a+1)$. Updating the posterior after a reward means adding one to a count: S_a after a success, F_a after a failure.

Algorithm 2.3 Thompson sampling

Input: number of arms K; a prior for each arm's mean ($\mathrm{Beta}(1,1)$ for Bernoulli rewards; $\mathcal{N}(m_0, s_0^2)$ for Gaussian rewards with known noise variance σ^2).

1. Set $S_a \leftarrow 0$ and $F_a \leftarrow 0$ for every arm (Gaussian case: $N_a \leftarrow 0$ and the reward sum $\Sigma_a \leftarrow 0$).
2. For $t = 1, 2, \ldots, n$:
   1. For every arm draw $\tilde{\theta}_a \sim \mathrm{Beta}(S_a+1, F_a+1)$ (Gaussian case: $\tilde{\theta}_a \sim \mathcal{N}(m_a, s_a^2)$ with $1/s_a^2 = 1/s_0^2 + N_a/\sigma^2$ and $m_a = s_a^2\,(m_0/s_0^2 + \Sigma_a/\sigma^2)$).
   2. Pull $a_t = \arg\max_a \tilde{\theta}_a$ and observe r_t.
   3. Bernoulli case: if $r_t = 1$, $S_{a_t} \leftarrow S_{a_t} + 1$, otherwise $F_{a_t} \leftarrow F_{a_t} + 1$. Gaussian case: $N_{a_t} \leftarrow N_{a_t} + 1$ and $\Sigma_{a_t} \leftarrow \Sigma_{a_t} + r_t$.

Output: the posteriors of all arms.

The Gaussian version uses the conjugate Normal prior for a mean when the noise variance σ^2 is known. Its posterior precision (one over the variance) is the prior precision plus N_a/σ^2: every pull adds the same amount of information. The posterior mean is a precision-weighted average of the prior mean and the sample average, which for large N_a is close to the sample average.

Example 2.5 Probability matching

Arm A has produced 6 successes and 4 failures, so its posterior is Beta$(7, 5)$ with mean $7/12 \approx 0.58$. Arm B has produced 1 success and 1 failure, so its posterior is Beta$(2, 2)$ with mean 0.5. A greedy agent would pull A. Thompson sampling pulls B whenever its draw is the larger one, which happens with probability

$$\Pr(\tilde{\theta}_B > \tilde{\theta}_A) = \int_0^1 p_B(x)\, \Pr(\tilde{\theta}_A < x)\, dx = \frac{5}{13} \approx 0.385,$$

where p_B is the density of Beta$(2, 2)$. (The integral can be evaluated exactly because the Beta parameters are integers; the tests of the lab check it numerically.) Arm B looks worse on average, but its posterior is so wide that it might well be the better arm, and Thompson sampling gives it more than a third of the pulls. As B is pulled, its posterior narrows, and if B really is worse its share of the pulls shrinks toward zero.

♦

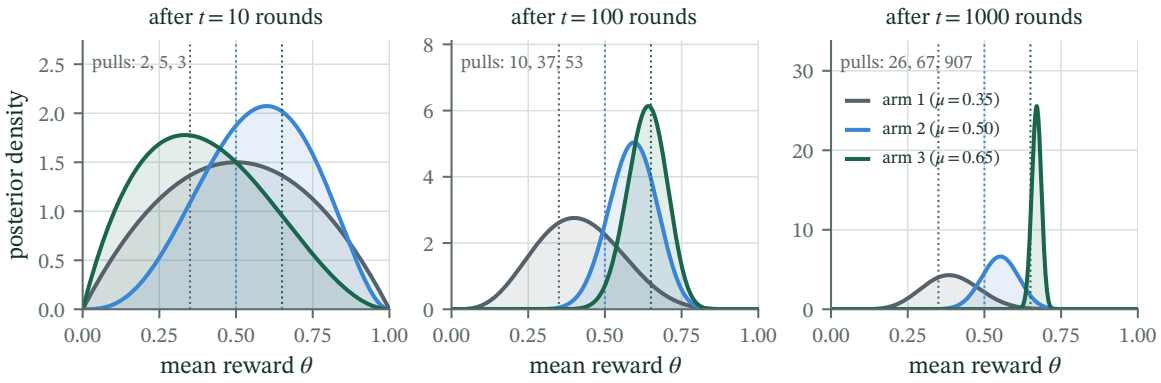


Figure 2.4. Thompson sampling's posteriors in one run on a three-armed Bernoulli bandit with means 0.35, 0.5 and 0.65 (dotted lines). After 10 rounds (left) the posteriors are wide and overlap. After 100 rounds the best arm has the most pulls. After 1000 rounds the best arm's posterior is a narrow spike near 0.65, while the posteriors of the worse arms stay wide: the agent stopped pulling them as soon as they were clearly worse, without measuring them precisely.

Figure 2.4 shows the posteriors evolving during one run. The picture is the Bayesian counterpart of Figure 2.2: uncertainty about the poor arms is never resolved, and it does not need to be. On the testbed (Figure 2.1), Gaussian Thompson sampling with prior $\mathcal{N}(0, 1)$, which happens to be exactly the distribution the testbed draws its means from, plays the optimal arm in 91% of the last 100 rounds. On the two-armed problem of Figure 2.3 its regret after 10^4 rounds is about 14, against 62 for UCB1.

Thompson sampling does not need its prior to be correct to work well: with the uniform prior on Bernoulli arms its regret is logarithmic whatever the true means are, and asymptotically it matches the best possible constant (Section 2.11). A prior that is badly wrong, for example one that is confidently pessimistic about the best arm, can slow it down considerably, and a model that is wrong (Gaussian noise assumed, heavy tails in reality) can mislead it. The tutorial by Russo et al. (2018) covers both the theory and the many practical variants.

> **KEY IDEA**
>
> UCB1 and Thompson sampling explore in the same spirit. Both pull an arm when, in the light of the data, it could plausibly be the best: UCB1 because its upper confidence bound is the highest, Thompson sampling because its posterior draw is. Both stop pulling an arm once the data show that it is worse.

2.6 Gradient bandits

All the methods so far estimate the value of each arm and derive their choices from the estimates. A different approach learns the *policy* directly: a probability distribution over arms that is adjusted, round by round, in the direction that increases the expected reward. This approach needs no value estimates at all, and it is the seed of the policy-gradient methods of Chapter 11.

2.6.1 A softmax policy

Give each arm a real-valued *preference* h_a and choose arms with the *softmax* probabilities

$$\pi(a) = \frac{e^{h_a}}{\sum_{b=1}^{K} e^{h_b}}.$$

Higher preferences mean higher probabilities, and only the differences between preferences matter: adding the same number to every h_a leaves π unchanged. The expected reward of one round under this policy is

$$J(h) = \sum_{a=1}^{K} \pi(a)\,\mu_a.$$

We would like to increase J by gradient ascent, $h \leftarrow h + \eta\,\nabla J(h)$. The difficulty is that J depends on the unknown means. The solution is to find a quantity that we *can* compute from one round of play and whose expectation is the gradient.

2.6.2 The gradient, and a sample of it

Differentiating the softmax gives $\partial\pi(a)/\partial h_b = \pi(a)\big(\mathbb{1}\{a=b\} - \pi(b)\big)$, where $\mathbb{1}\{a=b\}$ is 1 if $a=b$ and 0 otherwise. Therefore

$$\frac{\partial J}{\partial h_b} = \sum_a \mu_a\,\pi(a)\big(\mathbb{1}\{a=b\} - \pi(b)\big) = \pi(b)\big(\mu_b - J(h)\big). \tag{2.4}$$

In words: the preference of arm b should increase if the arm is better than the current average reward, and decrease if it is worse, in proportion to how often the arm is played. The middle

expression is an expectation over the arm $a \sim \pi$: it is the average of $\mu_a(\mathbb{1}\{a = b\} - \pi(b))$. Since the reward r_t of the arm actually pulled has mean μ_{a_t}, the quantity $r_t\,(\mathbb{1}\{a_t = b\} - \pi(b))$ is an unbiased sample of the gradient. We can also subtract a *baseline* $\bar{r}$ from the reward without changing the expectation, as the next proposition shows.

Proposition 2.4 The gradient-bandit update is unbiased

Let $a_t \sim \pi$, let r_t have mean μ_{a_t} given a_t, and let $\bar{r}$ be any number that does not depend on a_t or r_t. Then for every arm b,

$$\mathbb{E}\big[(r_t - \bar{r})\big(\mathbb{1}\{a_t = b\} - \pi(b)\big)\big] = \frac{\partial J}{\partial h_b}.$$

Proof. Conditioning on a_t replaces r_t by μ_{a_t}, so the expectation is $\sum_a \pi(a)(\mu_a - \bar{r})(\mathbb{1}\{a = b\} - \pi(b))$. The part with μ_a is $\partial J/\partial h_b$ by (2.4). The part with $\bar{r}$ is $\bar{r}\sum_a \pi(a)(\mathbb{1}\{a = b\} - \pi(b)) = \bar{r}\,(\pi(b) - \pi(b)) = 0$. ∎

This gives the *gradient bandit* algorithm. A good baseline is the average of the rewards received so far; using only the rounds *before* the current one makes sure the baseline does not depend on the current choice, as the proposition requires.

Algorithm 2.4 Gradient bandit

Input: number of arms K, step size $\eta > 0$.

1. Set $h_a \leftarrow 0$ for every arm and the baseline $\bar{r} \leftarrow 0$.
2. For $t = 1, 2, \dots, n$:
   1. Compute $\pi(a) = e^{h_a} / \sum_b e^{h_b}$ for every arm.
   2. Draw $a_t \sim \pi$, pull it and observe r_t.
   3. For every arm b: $h_b \leftarrow h_b + \eta\,(r_t - \bar{r})\big(\mathbb{1}\{a_t = b\} - \pi(b)\big)$.
   4. $\bar{r} \leftarrow \bar{r} + (r_t - \bar{r})/t$.

Output: the preferences h and the policy π.

Step 3 reads naturally. If the reward beats the baseline, the preference of the chosen arm goes up, by $\eta(r_t - \bar{r})(1 - \pi(a_t))$, and the preferences of all other arms go down, each by $\eta(r_t - \bar{r})\pi(b)$. If the reward falls short of the baseline, the opposite happens. Arms are reinforced or weakened by how their rewards compare with what was expected, which is the law of effect of Chapter 1 in mathematical form.

Example 2.6 One gradient-bandit step

Three arms have means $\mu = (1, 2, 3)$ and the preferences start at $h = (0, 0, 0)$, so $\pi = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ and $J = 2$. By (2.4) the gradient is $\nabla J = \frac{1}{3}(1 - 2,\ 2 - 2,\ 3 - 2) = (-\frac{1}{3}, 0, \frac{1}{3})$. Suppose the

agent pulls arm 3 and receives $r = 3.5$, the baseline is $\bar{r} = 2$, and $\eta = 0.1$. The update adds $0.1 \times 1.5 \times (-\frac{1}{3}, -\frac{1}{3}, \frac{2}{3}) = (-0.05, -0.05, 0.10)$ to h, and the new policy is $\pi \approx (0.316, 0.316, 0.367)$. One lucky round moved probability toward the best arm. Averaged over the random choice of arm and reward, the update is $\eta \nabla J$.

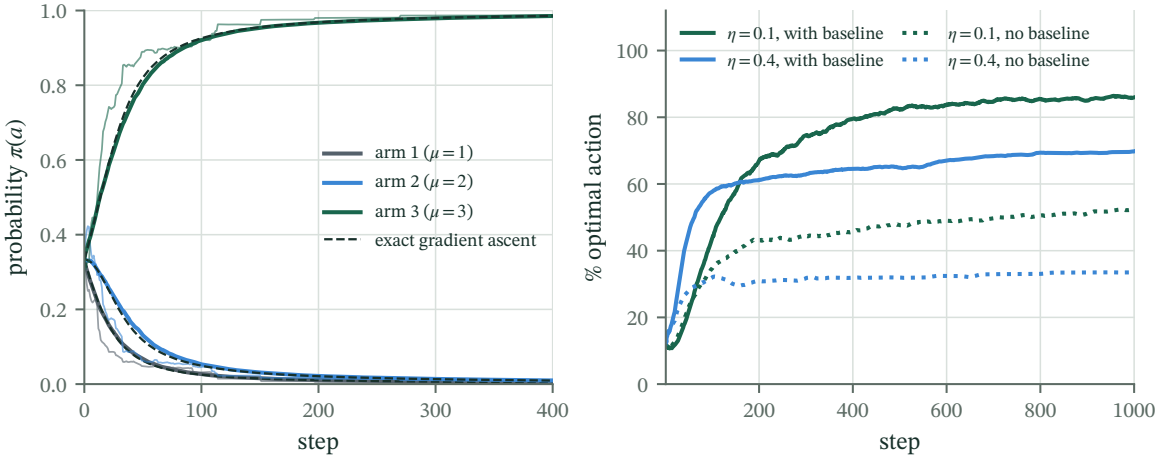


Figure 2.5. The gradient bandit. Left: the probabilities of three arms with means 1, 2 and 3 ($\eta = 0.1$, with baseline): one run (thin lines), the average of 200 runs (thick lines, with ± 1 standard-error bands) and exact gradient ascent on J (dashed), which the average follows closely. Right: percentage of optimal choices on a testbed whose means are drawn from $\mathcal{N}(4, 1)$, averaged over 200 problems. With all rewards near 4, the version without a baseline (dotted) learns much more slowly.

The left panel of Figure 2.5 shows the gradient bandit at work. The average over many runs stays close to the path of exact gradient ascent, as Proposition 2.4 suggests: on average, each update is a step of $\eta \nabla J$. Individual runs fluctuate around that path. The right panel shows why the baseline matters. When every reward is around 4, the update without a baseline pushes up the preference of whatever arm was chosen, good or bad, and only the small differences between these pushes carry information. The expected update is the same with or without the baseline, but its variance is much larger without one. Over the last 100 of 1000 rounds, the agent with a baseline picks the optimal arm 86% of the time with $\eta = 0.1$; without a baseline only 52%.

The update has another reading that is worth noticing now. The factor $\mathbb{1}\{a_t = b\} - \pi(b)$ is the derivative of $\ln \pi(a_t)$ with respect to h_b, so the update is $\eta\,(r_t - \bar{r})\,\nabla_h \ln \pi(a_t)$. This is the *score-function* form of a policy gradient, (11.4), and the gradient bandit is the one-step special case of the REINFORCE algorithm of Chapter 11, baseline included (Section 11.5).

PITFALL

The gradient bandit is sensitive to its step size and to the scale of the rewards, because both multiply the update. With $\eta = 0.4$ in Figure 2.5 it learns faster at first but settles less often on the best arm (70% with a baseline): a few unlucky large steps can make the policy nearly deterministic on a poor arm, after which the better arms are rarely tried and their preferences barely move. Chapter 14 studies when softmax policy gradients converge to the optimum and how slowly they can do so.

2.7 Bandits in practice

Table 2.2 summarizes the methods of this chapter.

Table 2.2. The bandit methods of this chapter at a glance.

Method	You must choose	How it explores	Main caveat
ε-greedy	ε or a schedule	random pulls	constant ε: linear regret
Optimistic start	initial value $\hat{\mu}_0$	inflated estimates	only at the start
UCB1	bonus scale c	bonus $\propto 1/\sqrt{N_a}$	needs the reward range
Thompson sampling	a prior and a reward model	posterior draws	needs a posterior
Gradient bandit	step size η, baseline	random choice from π	sensitive to η and scale

Which to use? When rewards are clicks, conversions or other successes and failures, Thompson sampling with Beta posteriors is simple, needs no tuning and is hard to beat; in an empirical study that included display advertising and news recommendation, Chapelle and Li (2011) found it competitive with or better than UCB-type methods. UCB is a good choice when you want a deterministic method with a guarantee that does not depend on a prior. ε-greedy is the universal baseline: it works with any estimates, including those of a neural network, which is why deep reinforcement learning uses it so often (Chapter 10). The gradient bandit matters less as a bandit algorithm than as the first policy-gradient method.

A few practical issues come up again and again.

Changing environments. If the arms drift, use constant step sizes or estimates over a sliding window. Thompson sampling can forget too: discount old successes and failures. In Lab Task 6, on a slowly drifting bandit, a constant step size of 0.1 earns 0.90 per round over the last 1000 rounds, against 0.69 for sample averages.

Delayed and batched feedback. In many systems rewards arrive late, or decisions are made in batches of thousands before any feedback returns. A deterministic rule such as UCB then makes the same choice for the whole batch. Randomized rules such as Thompson sampling spread a batch over all the arms that might be best, which is one reason they are popular in industry.

Bandits versus A/B tests. A classical A/B test splits traffic evenly and aims to measure the difference between options precisely. A bandit shifts traffic toward the better option during the experiment and aims at small regret. The two goals conflict: a bandit collects little data about poor arms, so its estimates of them are imprecise. Worse, averages of adaptively collected data are biased. An arm that was unlucky early is pulled less, so its low average is rarely corrected, and the averages tend to underestimate the means (Nie et al. 2018). If you need valid statistics at the end, design for them, for instance with the pure-exploration methods of Section 2.11.

PITFALL

Do not tune ε, c or η on the same problems you use to report performance: the tuned method will look better than it is. Tune on separate problems, report averages over many runs with their variability, and compare methods with equal tuning effort (Section 23.4).

LAB

Lab 2.1: Bandits on the testbed (`labs/ch02_bandits.py`, about 45 minutes). Run `python3 labs/ch02_bandits.py`. You will compare greedy and ε-greedy agents on the 10-armed testbed, tune ε, try optimistic initial values, measure the regret of UCB1 and Thompson sampling against the Lai–Robbins constant, see what the baseline does for the gradient bandit, track a drifting bandit with a constant step size, and run LinUCB on a contextual bandit. The algorithms are in `rlcourse/bandits.py`.

2.8 Regret analysis of UCB1 ●●

This section proves the guarantee of Section 2.4. The proof is short, and it shows where the logarithm comes from: to rule out, with confidence $1 - t^{-4}$, that an arm with gap Δ is the best, the agent needs about $\ln t / \Delta^2$ samples of it.

2.8.1 Concentration of averages

The only probabilistic tool we need is Hoeffding's inequality.

Lemma 2.5 Hoeffding's inequality

Let $X_1, \dots, X_s$ be independent random variables with values in an interval $[\ell, u]$ and common mean μ, and let $\bar{X} = \frac{1}{s}\sum_{i=1}^{s} X_i$. Then for every $c \geq 0$,

$$\Pr\big(\bar{X} \geq \mu + c\big) \leq \exp\Big(-\frac{2sc^2}{(u-\ell)^2}\Big) \quad \text{and} \quad \Pr\big(\bar{X} \leq \mu - c\big) \leq \exp\Big(-\frac{2sc^2}{(u-\ell)^2}\Big).$$

Theorem B.9 in Appendix B proves it. The idea is to bound the moment-generating function of a bounded variable, $\mathbb{E}[e^{\lambda(X-\mu)}] \leq e^{\lambda^2 (u-\ell)^2/8}$, multiply over the independent terms, and apply Markov's inequality to $e^{\lambda s(\bar{X}-\mu)}$ with the best λ.

There is a subtlety in applying the lemma to a bandit algorithm. After t rounds, arm a has been pulled $N_a(t)$ times, a random number that depends on the rewards themselves; its average is not an average of a fixed number of independent rewards. A standard device removes the difficulty. Imagine that before the game starts, nature draws for each arm an infinite sequence $X_{a,1}, X_{a,2}, \dots$ of independent rewards from that arm's distribution, and that the s-th pull of arm a returns $X_{a,s}$. This *reward table* produces exactly the same joint distribution of choices and rewards as the bandit of Definition 2.1. Let $\hat{\mu}_{a,s} = \frac{1}{s}\sum_{i=1}^{s} X_{a,i}$ be the average of the first s entries for arm a. For each *fixed* s this is an average of s independent rewards, so Hoeffding's inequality applies to it, and the algorithm's estimate after t rounds is $\hat{\mu}_{a,N_a(t)}$. The proof will apply the lemma to every possible value of the count and combine the results with a union bound.

2.8.2 The regret bound

Theorem 2.6 Regret of UCB1

Suppose the rewards of each arm are independent and identically distributed with values in $[0, 1]$. Then for every $n \geq K$, UCB1 (Algorithm 2.2) pulls every suboptimal arm a at most

$$\mathbb{E}[N_a(n)] \leq \frac{8 \ln n}{\Delta_a^2} + 1 + \frac{\pi^2}{3}$$

times in expectation, and therefore

$$\mathrm{Reg}_n \leq \sum_{a:\, \Delta_a > 0} \Big(\frac{8 \ln n}{\Delta_a} + \Big(1 + \frac{\pi^2}{3}\Big)\Delta_a \Big).$$

Proof. Fix a suboptimal arm a and an optimal arm a^*, and write $c_{t,s} = \sqrt{2 \ln t / s}$ for the bonus of an arm with s pulls when t rounds have been played. In round $t > K$, UCB1 compares the indices $\hat{\mu}_{b,N_b(t-1)} + c_{t-1,N_b(t-1)}$.

Step 1: counting. Let $\ell \geq 1$ be an integer, to be chosen later. Among the rounds in which arm a is pulled, at most ℓ (including its initial pull) can occur while $N_a(t-1) < \ell$. Every other such round has $N_a(t-1) \geq \ell$, so

$$N_a(n) \leq \ell + \sum_{t=K+1}^{n} \mathbb{1}\{a_t = a,\ N_a(t-1) \geq \ell\}.$$

Step 2: when can a suboptimal arm win? Suppose $a_t = a$, and let $s = N_{a^*}(t-1)$ and $s' = N_a(t-1)$. Then arm a's index is at least that of a^*:

$$\hat{\mu}_{a^*,s} + c_{t-1,s} \leq \hat{\mu}_{a,s'} + c_{t-1,s'}.$$

This inequality implies that at least one of three events occurs:

$$\text{(i) } \hat{\mu}_{a^*,s} \leq \mu^* - c_{t-1,s}, \qquad \text{(ii) } \hat{\mu}_{a,s'} \geq \mu_a + c_{t-1,s'}, \qquad \text{(iii) } \mu^* < \mu_a + 2c_{t-1,s'}.$$

Indeed, if none of them occurs, then $\hat{\mu}_{a^*,s} + c_{t-1,s} > \mu^* \geq \mu_a + 2c_{t-1,s'} > \hat{\mu}_{a,s'} + c_{t-1,s'}$, contradicting the inequality. In words: a suboptimal arm can win only if the optimal arm is underestimated, if the suboptimal arm is overestimated, or if the suboptimal arm's bonus is still larger than half its gap.

Step 3: enough pulls rule out (iii). Choose $\ell = \lceil 8 \ln n / \Delta_a^2 \rceil$. For $s' \geq \ell$ and $t \leq n$,

$$2c_{t-1,s'} = 2\sqrt{\frac{2 \ln(t-1)}{s'}} \leq 2\sqrt{\frac{2 \ln n}{\ell}} \leq 2\sqrt{\frac{2 \ln n\, \Delta_a^2}{8 \ln n}} = \Delta_a,$$

so (iii), which says $\Delta_a < 2c_{t-1,s'}$, cannot occur.

Step 4: (i) and (ii) are rare. The counts s and s' are random, but they lie between 1 and $t-1$. Bounding the indicator of the event by the sum over all possible values of the counts (a union bound),

$$\mathbb{1}\{a_t = a,\, N_a(t-1) \geq \ell\} \leq \sum_{s=1}^{t-1} \sum_{s'=\ell}^{t-1} \Big(\mathbb{1}\{\hat{\mu}_{a^*,s} \leq \mu^* - c_{t-1,s}\} + \mathbb{1}\{\hat{\mu}_{a,s'} \geq \mu_a + c_{t-1,s'}\} \Big).$$

For fixed s, Lemma 2.5 gives $\Pr(\hat{\mu}_{a^*,s} \leq \mu^* - c_{t-1,s}) \leq \exp(-2s \cdot 2\ln(t-1)/s) = (t-1)^{-4}$, and the same bound holds for the second event. There are at most $(t-1)^2$ pairs (s, s'), so taking expectations in Step 1,

$$\mathbb{E}[N_a(n)] \leq \ell + \sum_{t=K+1}^{n} 2(t-1)^2 (t-1)^{-4} \leq \ell + 2 \sum_{m=1}^{\infty} \frac{1}{m^2} = \ell + \frac{\pi^2}{3}.$$

Since $\ell \leq 8\ln n/\Delta_a^2 + 1$, this is the first claim. The second follows from the regret decomposition (2.1). ■

The proof shows the three ways a suboptimal arm gets pulled. Bad luck on the optimal arm and good luck on the suboptimal one are made rare by the confidence level t^{-4} hidden in the bonus; their total contribution is a constant, $\pi^2/3$, that does not grow with n. The third way, lack of data, ends once arm a has about $8\ln n/\Delta_a^2$ pulls. The logarithm is the price of high confidence, and the $1/\Delta_a^2$ is the number of samples needed to resolve a difference of Δ_a in the presence of noise of constant size. Exercise 2.12 repeats the argument with a different bonus and shows how the constants trade off.

The bound depends on the gaps, and it becomes useless when a gap is tiny: $8\ln n/\Delta_a$ is huge, although an arm with a tiny gap costs almost nothing per pull. A bound that holds uniformly over all bandits follows by treating small and large gaps separately.

Corollary 2.7 A gap-free bound for UCB1

Under the assumptions of Theorem 2.6, for every $n \geq K$,

$$\mathrm{Reg}_n \leq 4\sqrt{2Kn\ln n} + \Big(1 + \frac{\pi^2}{3}\Big) \sum_{a=1}^{K} \Delta_a.$$

Proof. Fix a threshold $\Delta > 0$. Arms with $\Delta_a < \Delta$ contribute at most $\Delta \sum_a \mathbb{E}[N_a(n)] \leq n\Delta$ to the decomposition (2.1). Each arm with $\Delta_a \geq \Delta$ contributes at most $8\ln n/\Delta_a + (1+\pi^2/3)\Delta_a \leq 8\ln n/\Delta + (1+\pi^2/3)\Delta_a$ by the theorem. Hence $\mathrm{Reg}_n \leq n\Delta + 8K\ln n/\Delta + (1+\pi^2/3)\sum_a \Delta_a$. The choice $\Delta = \sqrt{8K\ln n/n}$ makes the first two terms equal to $\sqrt{8Kn\ln n}$ each, and $2\sqrt{8Kn\ln n} = 4\sqrt{2Kn\ln n}$. ■

So UCB1's worst-case regret grows like $\sqrt{Kn\ln n}$. Section 2.9 shows that no algorithm can do better than order $\sqrt{Kn}$ in the worst case, so UCB1 is off by at most a factor of order $\sqrt{\ln n}$.

2.8.3 Explore-then-commit

A simpler strategy separates exploration from exploitation completely: pull every arm m times, then commit to the arm with the best average for the remaining rounds. *Explore-then-commit* is what a careful experimenter does when running an A/B test before a launch. How good is it?

Proposition 2.8 Explore-then-commit with two arms

Consider a bandit with two arms, rewards in $[0,1]$ and gap $\Delta > 0$. Explore-then-commit pulls each arm m times, with $2m \le n$, and then pulls the arm with the larger average in all remaining rounds, breaking ties arbitrarily. Its regret satisfies

$$\mathrm{Reg}_n \le m\Delta + (n-2m)\,\Delta\, e^{-m\Delta^2/2}.$$

Proof. During exploration the suboptimal arm is pulled m times, which costs $m\Delta$. Afterwards each round costs Δ if the agent committed to the wrong arm, which can happen only if the suboptimal arm's average is at least the optimal arm's. In the reward table, that event says that the average of the m differences $X_{2,i} - X_{1,i}$, where arm 1 is optimal, is at least 0. These differences are independent, lie in $[-1,1]$ and have mean $-\Delta$, so by Lemma 2.5 with $u-\ell = 2$ the probability is at most $\exp(-2m\Delta^2/4) = e^{-m\Delta^2/2}$. ■

The two terms pull in opposite directions: more exploration costs more now and errs less later. Minimizing the bound over m gives $m \approx (2/\Delta^2)\ln(n\Delta^2/2)$ and regret of order $(\ln n)/\Delta$, the same order as UCB1 (Exercise 2.13). On the two-armed bandit with means 0.4 and 0.6 and $n = 10^4$, the tuned value is $m = 265$, and explore-then-commit's regret is about 53, a little better than UCB1's 62. But there is a catch: the best m depends on the gap Δ and on the horizon n, which are usually unknown. If m is chosen without knowing Δ, for instance $m \approx n^{2/3}$, then for some gaps the regret is of order $n^{2/3}$ (Exercise 2.13), much worse than UCB1's worst case of order $\sqrt{n \ln n}$. UCB1 achieves the right amount of exploration for every gap at once, without being told.

2.9 Lower bounds ●●

How good can a bandit algorithm be? The question needs care. No algorithm can be best on every bandit: the algorithm that always pulls arm 1 has zero regret on every bandit in which arm 1 happens to be optimal. To rule out such "lucky guessers" we compare algorithms that work reasonably well on *all* bandits of a class.

Definition 2.9 Consistent algorithm

An algorithm is *consistent* over a class of bandits if, on every bandit of the class and for every $\alpha > 0$, its regret satisfies $\mathrm{Reg}_n / n^\alpha \to 0$ as $n \to \infty$.

UCB1 and Thompson sampling are consistent over Bernoulli bandits, since their regret grows logarithmically; ε-greedy with a constant ε and the lucky guesser are not.

2.9.1 The Lai–Robbins bound

For Bernoulli arms, write $\mathrm{kl}(p,q) = p\ln\frac{p}{q} + (1-p)\ln\frac{1-p}{1-q}$ for the Kullback–Leibler divergence between Bernoulli distributions with means p and q (Appendix A). It measures how easily samples from the first distribution can be told apart from samples of the second.

Theorem 2.10 Lai–Robbins lower bound

Consider the class of bandits whose arms are Bernoulli with means in $(0,1)$. For every algorithm that is consistent over this class, every bandit in it and every suboptimal arm a,

$$\liminf_{n\to\infty} \frac{\mathbb{E}[N_a(n)]}{\ln n} \geq \frac{1}{\mathrm{kl}(\mu_a,\mu^*)}, \qquad \text{hence} \qquad \liminf_{n\to\infty} \frac{\mathrm{Reg}_n}{\ln n} \geq \sum_{a:\,\Delta_a>0} \frac{\Delta_a}{\mathrm{kl}(\mu_a,\mu^*)}.$$

Lai and Robbins (1985) proved this for families of reward distributions with one real parameter, under regularity conditions that the Bernoulli family and the Gaussian family with known variance satisfy; for those, $\mathrm{kl}(\mu_a,\mu^*)$ becomes the KL divergence between the distributions of arm a and of an optimal arm. Burnetas and Katehakis (1996) extended it to more general classes. The theorem says that logarithmic regret is not just achievable but unavoidable, and it gives the best possible constant.

Why it holds. To be confident that arm a is not the best, the agent must rule out the possibility that a's mean is actually slightly larger than μ^*. That alternative bandit differs from the true one only in arm a, so only pulls of arm a carry information to tell them apart, about $\mathrm{kl}(\mu_a,\mu^*)$ nats per pull. And the agent must tell them apart with error probability about $1/n$, because mistaking the true bandit for the alternative, or the reverse, would cost regret proportional to n. Distinguishing two hypotheses with error probability $1/n$ takes about $\ln n$ divided by the information per sample.

Proof sketch of the Lai–Robbins bound. Fix a consistent algorithm, a bandit ν and a suboptimal arm a, and let ν' be the bandit that agrees with ν except that arm a has mean $\mu'_a > \mu^*$, so that a is the only optimal arm of ν'. Write $\Pr$ and $\Pr'$ for the distributions of the n rounds of choices and rewards under ν and ν'. Two facts are needed (Lattimore and Szepesvári 2020, ch. 14–16). First, because the bandits differ only in arm a, the KL divergence between the two distributions of the whole history is $D_{\mathrm{KL}}(\Pr \,\|\, \Pr') = \mathbb{E}[N_a(n)]\,\mathrm{kl}(\mu_a,\mu'_a)$. Second, the Bretagnolle–Huber inequality says that for every event E, $\Pr(E) + \Pr'(E^c) \geq \frac{1}{2}\exp(-D_{\mathrm{KL}}(\Pr \,\|\, \Pr'))$.

Take $E = \{N_a(n) > n/2\}$. Under ν, Markov's inequality and the regret decomposition give $\Pr(E) \leq 2\mathbb{E}[N_a(n)]/n \leq 2\mathrm{Reg}_n(\nu)/(n\Delta_a)$. Under ν' every arm other than a has gap at least $\mu'_a - \mu^*$, so $\Pr'(E^c) \leq 2\mathrm{Reg}_n(\nu')/(n(\mu'_a - \mu^*))$. By consistency, both regrets are at most Cn^α for any fixed $\alpha > 0$, some constant C and all large n. Hence $\frac{1}{2}\exp(-\mathbb{E}[N_a(n)]\,\mathrm{kl}(\mu_a,\mu'_a)) \leq C'n^{\alpha-1}$, that is,

$$\mathbb{E}[N_a(n)] \geq \frac{(1-\alpha)\ln n - \ln(2C')}{\mathrm{kl}(\mu_a,\mu'_a)}.$$

Divide by $\ln n$ and let $n \to \infty$, then let $\alpha \to 0$ and $\mu'_a \downarrow \mu^*$; since $\mathrm{kl}(\mu_a,\cdot)$ is continuous, the bound tends to $1/\mathrm{kl}(\mu_a,\mu^*)$. ■

Some numbers make the theorem concrete. For the two-armed bandit with means 0.4 and 0.6, $\mathrm{kl}(0.4, 0.6) = 0.081$, and the Lai–Robbins constant is $0.2/0.081 = 2.47$. UCB1's bound has the constant $8/\Delta = 40$ instead, more than sixteen times larger. Pinsker's inequality $\mathrm{kl}(p, q) \geq 2(p - q)^2$ shows that the Lai–Robbins constant of an arm is at most $1/(2\Delta_a)$, with near-equality when the means are close to $\frac{1}{2}$; for arms near 0 or 1, whose rewards are less noisy, it can be much smaller. With means 0.01 and 0.11 the constant is 1.23, against $1/(2\Delta) = 5$. An algorithm based on Hoeffding's inequality, which assumes the worst-case variance $\frac{1}{4}$ for every arm, cannot exploit this; KL-UCB (Section 2.11) can.

Remember that the theorem is asymptotic. The right panel of Figure 2.3 plots $\mathrm{Reg}_n / \ln n$: Thompson sampling's ratio is still well below 2.47 at $n = 10^4$ and creeps upward. A finite-time regret below the constant times $\ln n$ does not contradict the theorem, which constrains only the limit.

2.9.2 The minimax bound

The Lai–Robbins bound depends on the bandit. A complementary question asks for the best guarantee that holds for all bandits at once.

Theorem 2.11 Minimax lower bound

There is a universal constant $c > 0$ such that for every $K \geq 2$, every $n \geq K$ and every algorithm, some K-armed bandit with Bernoulli rewards has $\mathrm{Reg}_n \geq c\sqrt{Kn}$.

The theorem is due to Auer, Cesa-Bianchi, Freund, et al. (2002); Lattimore and Szepesvári (2020, ch. 15) give a proof for Gaussian rewards with an explicit constant. The hard bandits are easy to describe: all arms have mean $\frac{1}{2}$ except one, which has mean $\frac{1}{2} + \varepsilon$ with ε of order $\sqrt{K/n}$. In n rounds the agent can pull each arm only about n/K times, too few to detect a difference of $\sqrt{K/n}$, so it cannot find the better arm reliably and loses about ε per round, $\varepsilon n \approx \sqrt{Kn}$ in total.

These two bounds give two meanings to "optimal". An algorithm is *asymptotically optimal* if it attains the Lai–Robbins constant on every bandit of the class, and *minimax optimal* if its worst-case regret is of order $\sqrt{Kn}$. UCB1 is neither exactly: its constant is too large and its worst case has an extra $\sqrt{\ln n}$. The MOSS algorithm of Audibert and Bubeck (2009) is minimax optimal; KL-UCB and Thompson sampling are asymptotically optimal for Bernoulli arms; and Ménard and Garivier (2017) gave an algorithm that is both.

2.10 Contextual bandits ●●

Real decisions usually come with information. A news site knows something about the reader before choosing an article; a doctor knows the patient's age and test results. The best arm depends on this *context*, and an algorithm that ignores it can do no better than the best single arm for the average reader.

In a *contextual bandit*, each round begins with the agent observing a context. It chooses an arm and receives a reward whose distribution depends on both the context and the arm. The next

context arrives independently of the agent's choice. This sits between the bandit and the full reinforcement-learning problem: there are states (the contexts), but actions do not influence which state comes next. It is reinforcement learning with episodes of length one.

2.10.1 Linear rewards

With many possible contexts the agent cannot learn a separate bandit for each one; it must generalize. The most studied model is linear. In round t, each arm a comes with a feature vector $x_{t,a} \in \mathbb{R}^d$ that describes the arm in the current context (for example, features of the reader combined with features of the article), and the expected reward is

$$\mathbb{E}[r_t \mid a_t = a] = \theta^{*\top} x_{t,a}$$

for an unknown parameter vector $\theta^* \in \mathbb{R}^d$. Here, too, θ^* describes the environment, not a policy. After t rounds, the agent estimates θ^* by *ridge regression* on the features $x_s = x_{s,a_s}$ of the arms it pulled:

$$\hat{\theta} = V^{-1} b, \qquad V = \lambda I + \sum_{s=1}^{t} x_s x_s^\top, \qquad b = \sum_{s=1}^{t} r_s x_s,$$

where $\lambda > 0$ is a regularization constant. The matrix V records how much information the data contain in each direction of feature space: a direction in which many pulled arms had large components is well explored.

Optimism now takes a geometric form. The plausible values of θ^* form an ellipsoid $\{\theta : \|\theta - \hat{\theta}\|_V \le \alpha\}$ around the estimate, where $\|y\|_V = \sqrt{y^\top V y}$. The most optimistic value of arm a's reward over this ellipsoid is

$$\max_{\|\theta - \hat{\theta}\|_V \le \alpha} \theta^\top x_{t,a} = \hat{\theta}^\top x_{t,a} + \alpha \sqrt{x_{t,a}^\top V^{-1} x_{t,a}},$$

by the Cauchy–Schwarz inequality (Exercise 2.15). The bonus $\alpha \|x_{t,a}\|_{V^{-1}}$ is large for feature directions that the data have not yet explored. Pulling one arm therefore reduces the uncertainty of every arm with similar features: the algorithm explores directions, not arms.

Algorithm 2.5 LinUCB

Input: regularization $\lambda > 0$, exploration strength $\alpha \ge 0$.

1. $V \leftarrow \lambda I$ (a $d \times d$ matrix), $b \leftarrow 0 \in \mathbb{R}^d$.
2. For $t = 1, 2, \ldots, n$:
   1. Observe the feature vectors $x_{t,a}$ of all arms.
   2. $\hat{\theta} \leftarrow V^{-1} b$.
   3. For every arm: $U_a \leftarrow \hat{\theta}^\top x_{t,a} + \alpha \sqrt{x_{t,a}^\top V^{-1} x_{t,a}}$.
   4. Pull $a_t = \arg\max_a U_a$ and observe r_t.
   5. $V \leftarrow V + x_{t,a_t} x_{t,a_t}^\top$ and $b \leftarrow b + r_t\, x_{t,a_t}$.

Output: the estimate $\hat{\theta}$.

This version, with one parameter vector shared by all arms, is the setting analyzed by Chu et al. (2011), who prove their regret bound for a modified version of the algorithm (SupLinUCB). Li et al. (2010) introduced LinUCB for news recommendation on the Yahoo! front page, with a separate parameter vector per article (the "disjoint" model, the special case of block-structured features) and a hybrid model that also shares parameters across articles, and reported a 12.5% higher click rate than a bandit algorithm that ignored context. Evaluating a bandit algorithm without deploying it is itself a problem; their evaluation replayed logged data collected by a uniformly random policy (Li et al. 2011), a simple instance of the off-policy evaluation methods of Chapter 17.

Lab Task 7 shows that how much exploration a contextual bandit needs depends on the contexts. When every round brings fresh, diverse feature vectors, even the greedy choice $\alpha = 0$ learns well, because the variety of contexts explores for free; Bastani et al. (2021) analyze when this happens. When the arms' features are the same in every round, the problem is a plain linear bandit, and greedy often commits to a poor arm: in the lab its regret after 3000 rounds is about 30 times that of LinUCB with $\alpha = 1$.

RESEARCH NOTE

Theory of linear bandits (●●●). The right size of α comes from a confidence set for ridge regression. If the noise is conditionally R-subgaussian and $\|\theta^*\|_2 \le S$, then with probability at least $1-\delta$, simultaneously for all t, $\|\hat{\theta}_t - \theta^*\|_{V_t} \le R\sqrt{2\ln\!\left(\det(V_t)^{1/2}\det(\lambda I)^{-1/2}/\delta\right)} + \sqrt{\lambda}\,S$ (Abbasi-Yadkori et al. 2011). Using this radius gives the OFUL algorithm, with regret $\tilde{O}(d\sqrt{n})$, where $\tilde{O}$ hides logarithmic factors; for some action sets no algorithm does better than order $d\sqrt{n}$ (Dani et al. 2008; Lattimore and Szepesvári 2020, ch. 24). Thompson sampling has a linear version whose regret bound is within a factor of about $\sqrt{d}$ of OFUL's (Agrawal and Goyal 2013). For general function classes, contextual bandits reduce to regression (Foster and Rakhlin 2020). Contextual bandits are also one-step reinforcement learning, and that view returns in Section 22.1: a language model answering a prompt, with a reward for the whole answer, faces a contextual bandit whose context is the prompt and whose arms are all possible responses.

2.11 Beyond the basic bandit ●●●

Bandit theory is a large field with its own textbooks (Lattimore and Szepesvári 2020; Slivkins 2019). This section points to its main directions.

Asymptotically optimal algorithms. The gap between UCB1's constant and the Lai–Robbins constant comes from Hoeffding's inequality, which treats every arm as if it had the largest possible variance. *KL-UCB* replaces the Hoeffding interval by a KL ball: the index of arm a is the largest q with $N_a\,\mathrm{kl}(\hat{\mu}_a, q) \le \ln t + c\ln\ln t$. For Bernoulli rewards its regret satisfies $\limsup_n \mathbb{E}[N_a(n)]/\ln n \le 1/\mathrm{kl}(\mu_a, \mu^*)$, matching Theorem 2.10 (Garivier and Cappé 2011); the analysis uses $c = 3$, and $c = 0$ is the usual choice in practice. Thompson sampling was long used

without guarantees. Agrawal and Goyal (2012) proved the first logarithmic regret bound for it, and Kaufmann et al. (2012) showed that with Beta priors it attains the Lai–Robbins constant for Bernoulli bandits. In a Bayesian sense, Thompson sampling inherits the guarantees of UCB algorithms: any regret bound proved for a UCB algorithm with valid confidence sets also bounds the Bayesian regret of Thompson sampling (Russo and Van Roy 2014), which helps explain its strong performance in structured problems.

Adversarial bandits. Drop the assumption that rewards are random draws: an adversary fixes an arbitrary table of rewards $x_{t,a} \in [0,1]$ in advance, and regret is measured against the best single arm in hindsight, $\max_a \sum_t x_{t,a} - \mathbb{E}[\sum_t x_{t,a_t}]$. Deterministic algorithms such as UCB1 fail, because the adversary can anticipate them; the agent must randomize. EXP3 (exponential weights for exploration and exploitation) keeps an importance-weighted estimate $\hat{S}_a$ of each arm's total reward, made unbiased by dividing what is observed about the pulled arm by the probability of having pulled it, and plays arm a with probability proportional to $e^{\eta \hat{S}_a}$. With a suitable step size its regret is $O(\sqrt{nK \ln K})$ against every reward table (Auer, Cesa-Bianchi, Freund, et al. 2002; Lattimore and Szepesvári 2020, ch. 11), within a $\sqrt{\ln K}$ factor of the minimax bound. The adversarial view connects bandits to online learning and game theory, where no-regret algorithms find equilibria (Chapter 21); `rlcourse.bandits.EXP3` implements it.

Pure exploration. Sometimes the goal is not to earn reward during the experiment but to identify the best arm at its end, for example before launching a product. In *best-arm identification* with fixed confidence, the agent stops as soon as it can name the best arm with error probability at most δ; it needs a number of samples of order $\sum_a \Delta_a^{-2} \ln(1/\delta)$ up to logarithmic factors (Even-Dar et al. 2006), and Garivier and Kaufmann (2016) characterized the optimal constant. Low regret and good identification are in tension: the smaller an algorithm's cumulative regret, the more slowly its recommendation of the best arm can improve (Bubeck et al. 2009).

Bayesian optimality. If the agent has a prior over the arms and maximizes the expected discounted reward $\mathbb{E}[\sum_t \gamma^{t-1} r_t]$, the problem is a Markov decision process whose states are posteriors: a *belief MDP* (Section 19.2.1). Its optimal policy trades off exploration and exploitation exactly. Gittins (1979) showed that when the arms are independent, this policy has a remarkably simple form: compute for each arm separately an *index* that depends only on that arm's posterior and on γ, and pull the arm with the largest index (Gittins et al. 2011). The index of an arm is the constant reward per round that would make the agent indifferent between that sure reward forever and playing the arm optimally with the option to switch to the sure reward later. For a Beta$(1,1)$ arm and $\gamma = 0.9$ it is about 0.70, well above the arm's posterior mean 0.5: the difference is the value of the information the arm may provide (Exercise 2.17). The Gittins index is optimal only under its assumptions (independent arms, geometric discounting, a correct prior); Thompson sampling and UCB give up exact Bayesian optimality in exchange for simplicity and guarantees that need no prior.

Non-stationary bandits. When the means can change, the comparison is with the best arm *at each time*, and regret must grow faster than in the stationary case. For piecewise-stationary bandits with Γ change points, UCB with a sliding window or with discounted counts has regret of order $\sqrt{\Gamma n}$ up to logarithmic factors when Γ is known in advance (Garivier and Moulines 2011). Adversarial algorithms such as EXP3 are a different defense: they make no stationarity

assumption at all.

RESEARCH NOTE

Open directions include: bandits with very large or continuous action sets and the structural assumptions that make them learnable; algorithms that are simultaneously optimal for stochastic and adversarial rewards ("best of both worlds"); delayed, batched and privacy-constrained feedback; valid statistical inference after adaptive experiments; and the step from one-state exploration to deep exploration in sequential problems, the subject of Chapter 16.

2.12 Summary

- A K-armed bandit is the simplest reinforcement-learning problem: one state, repeated choices, unknown and noisy rewards. Performance is measured by the regret, the expected reward lost compared with always playing the best arm, which decomposes as $\mathrm{Reg}_n = \sum_a \Delta_a \mathbb{E}[N_a(n)]$.
- Action values are estimated by the incremental update $\hat{\mu} \leftarrow \hat{\mu} + \eta\,(r - \hat{\mu})$: sample averages with $\eta = 1/N$, exponential recency-weighted averages with a constant η for changing problems. This "old estimate plus step size times error" form recurs throughout the book.
- Greedy selection can lock onto a poor arm forever. ε-greedy selection always explores, but blindly and forever, so a constant ε has linear regret. Optimistic initial values make even greedy agents explore, but only at the start.
- Upper confidence bounds add a bonus $\sqrt{2 \ln t / N_a}$ that shrinks as an arm is pulled: optimism in the face of uncertainty. UCB1 has logarithmic regret, at most $\sum_a (8 \ln n/\Delta_a + (1 + \pi^2/3)\Delta_a)$ for rewards in $[0, 1]$.
- Thompson sampling draws a plausible mean for every arm from its posterior and plays the best draw; it pulls each arm with the posterior probability that the arm is optimal, and it often has the smallest regret in practice.
- The gradient bandit learns a softmax policy by stochastic gradient ascent on the expected reward; a baseline leaves the expected update unchanged and reduces its variance. It is the first policy-gradient method of the book.
- (●●) No consistent algorithm can have regret below $\sum_a \Delta_a \ln n / \mathrm{kl}(\mu_a, \mu^*)$ asymptotically (Lai–Robbins), and every algorithm has worst-case regret of order at least $\sqrt{Kn}$. Contextual bandits add features, and LinUCB explores along the directions of feature space that the data have not covered.

Bandits ignore the fact that actions change the situation the agent is in. The next chapter introduces Markov decision processes, the model of sequential decisions in which they do.

2.13 Further Reading

Sutton and Barto (2018, ch. 2) treats the methods of the ● sections from the reinforcement-learning perspective, with the 10-armed testbed and many exercises. The definitive modern reference is Lattimore and Szepesvári (2020), which covers stochastic, adversarial, contextual

and linear bandits with complete proofs; its early chapters are the best next step after Section 2.8 and 2.9. Slivkins (2019) is a shorter and gentler textbook, and Bubeck and Cesa-Bianchi (2012) a concise survey of regret analysis in both the stochastic and the adversarial settings.

The original papers remain readable. Thompson (1933) introduced posterior sampling for clinical trials; Robbins (1952) posed the sequential design problem; Lai and Robbins (1985) proved the lower bound and constructed the first asymptotically optimal algorithms; Auer, Cesa-Bianchi, and Fischer (2002) introduced UCB1 with its finite-time analysis, which is the template of countless later proofs; Auer, Cesa-Bianchi, Freund, et al. (2002) introduced EXP3 and the adversarial bandit. Gittins (1979) is the classic paper on Bayesian optimality, and Gittins et al. (2011) the book-length treatment.

For Thompson sampling, the tutorial by Russo et al. (2018) explains the algorithm, its variants and its theory with many examples, and Chapelle and Li (2011) is the empirical study that revived interest in it. The asymptotically optimal analyses are Garivier and Cappé (2011) for KL-UCB and Kaufmann et al. (2012) for Thompson sampling.

For contextual bandits, Li et al. (2010) is the paper that brought LinUCB to news recommendation, and Abbasi-Yadkori et al. (2011) gives the confidence sets that underlie most of the linear-bandit theory. (●●●) For pure exploration, start with Even-Dar et al. (2006) and Garivier and Kaufmann (2016); for the interplay between regret and inference, Nie et al. (2018).

2.14 Exercises

Exercise 2.1 ●● Regret, realized and in hindsight

(a) Prove the regret decomposition (2.1), saying exactly where you use the fact that a pulled arm's reward is a fresh draw, independent of the past. (b) The *realized regret* is $n\mu^* - \sum_t r_t$. Show that it can be negative and that its expectation is Reg_n. (c) A stricter benchmark is the best arm *in hindsight*: suppose, as in the reward-table device of Section 2.8 but indexed by rounds, that the reward $X_{t,a}$ that arm a would give in round t is fixed in advance, and compare with $\mathbb{E}[\max_a \sum_{t=1}^{n} X_{t,a}]$. Show that this benchmark is at least $n\mu^*$. For two Bernoulli arms with mean $\frac{1}{2}$ each, show that every algorithm has $\mathrm{Reg}_n = 0$ but falls short of the hindsight benchmark by about $\sqrt{n/\pi}/2$ (use the normal approximation), and compute the shortfall for $n = 100$.

Exercise 2.2 ● The incremental mean

(a) Derive the incremental form (2.2) of the sample average. (b) Show that with step sizes $\eta = 1/N$ the estimate after one or more rewards does not depend on the initial estimate. (c) An arm returns the rewards $2, 4, 3, 7$. Compute the estimates after each reward with step sizes $1/N$, and with the constant step size 0.5 starting from 0.

Exercise 2.3 ● Exponential recency weights

(a) Derive (2.3) by induction on n. (b) Show that the weights $(1-\eta)^n$ and $\eta(1-\eta)^{n-i}$, $i = 1, \dots, n$, add up to one. (c) With $\eta = 0.1$, after how many rewards does the weight of the initial estimate drop below 1%? (d) For large n, what total weight do the rewards that are more than 20 pulls old receive?

Exercise 2.4 • When does greed fail?

Generalize Example 2.3 to Bernoulli arms with means $0 < \mu_A < \mu_B \leq 1$. (a) Show that the greedy agent locks onto arm A with probability $\mu_A/(\mu_A + \mu_B)$. (b) Find its long-run regret per round. (c) For $\mu_B = 0.9$, which μ_A makes this per-round regret largest? (d) Explain why the analysis breaks down if the initial estimates are 1 instead of 0.

Exercise 2.5 • How often does ε-greedy pull the best arm?

(a) Show that once the greedy arm is an optimal arm (and stays so), an ε-greedy agent pulls an optimal arm with probability $1 - \varepsilon + \varepsilon/K$ (assume a unique optimal arm). (b) Show that its regret per round then is $\frac{\varepsilon}{K}\sum_a \Delta_a$. (c) On the 10-armed testbed the expected largest mean is about 1.539 and the expected sum of the means is 0. Predict the long-run average reward for $\varepsilon = 0.1$ and $\varepsilon = 0.01$, and compare with Figure 2.1. Why is $\varepsilon = 0.01$ still behind after 1000 rounds?

Exercise 2.6 • Optimism on two arms

Two arms have deterministic rewards: arm 1 always pays 0.5 and arm 2 always pays 0.9. A greedy agent uses the constant step size $\eta = 0.5$ and breaks ties in favor of arm 1. (a) With initial estimates 0, what does it do? (b) With initial estimates 2, trace the first six rounds (arm chosen and both estimates). How many times does it pull arm 1, and when does it settle on arm 2? (c) For which initial values $\hat{\mu}_0$ does the agent eventually settle on arm 2? (d) Explain why optimism is less reliable when rewards are noisy.

Exercise 2.7 • A confidence radius from Hoeffding

Rewards lie in $[0, 1]$. (a) How many pulls guarantee, by Hoeffding's inequality, that the sample average underestimates the mean by more than 0.05 with probability at most 0.05? (b) How many guarantee that it misses the mean by more than 0.05 in either direction with probability at most 0.05? (c) For a Bernoulli arm with mean 0.5, compare with the numbers the normal approximation suggests, and explain the difference.

Exercise 2.8 • Thompson sampling by hand

An arm with prior Beta(1, 1) returns the rewards $1, 1, 0, 1, 0$. (a) What is its posterior, and what is the posterior mean? (b) What is the probability that the next pull returns 1, according to the posterior? (c) A second arm has never been pulled. With what probability does Thompson

sampling choose it over the first arm? (*Hint:* if U is uniform on $[0,1]$, then $\Pr(U > \theta) = 1 - \theta$.) (d) Redo (a) with the prior Beta(10, 10) and explain what this prior expresses.

Exercise 2.9 ●● The gradient of the expected reward

(a) Show that $\partial \pi(a)/\partial h_b = \pi(a)(\mathbb{1}\{a = b\} - \pi(b))$ for the softmax policy. (b) Derive (2.4). (c) Compute ∇J for $\mu = (1, 2, 3)$ and $h = (0, 0, \ln 2)$. (d) Show that the components of ∇J always sum to zero, and explain why in terms of the softmax.

Exercise 2.10 ●● What the baseline buys

Two arms have deterministic rewards $\mu_1 = 4$ and $\mu_2 = 5$, and the gradient bandit's policy is $\pi = (\frac{1}{2}, \frac{1}{2})$. Consider the update direction $g = (r - B)(\mathbb{1}\{a = 1\} - \pi(1))$ for h_1, with a constant baseline B. (a) Show that $\mathbb{E}[g]$ does not depend on B and equals $\partial J/\partial h_1$. (b) Compute $\mathrm{Var}(g)$ for $B = 0$ and for $B = J = 4.5$. (c) For a general bandit and policy, show that the constant baseline that minimizes $\mathrm{Var}(g)$ is $B^* = \mathbb{E}[r\,w^2]/\mathbb{E}[w^2]$ with $w = \mathbb{1}\{a = 1\} - \pi(1)$, and evaluate it here.

Exercise 2.11 ● Tracking a drifting bandit

Extend Lab Task 6. On the drifting bandit (ten arms, all means starting at 0, random-walk steps of standard deviation 0.01, rewards with unit noise), run ε-greedy with $\varepsilon = 0.1$ for 5000 rounds using sample averages and constant step sizes $\eta \in \{0.01, 0.03, 0.1, 0.3\}$, and measure the average reward over the last 1000 rounds (average over 100 runs). Which step size is best? Explain the trade-off with (2.3) and the variance formula of Section 2.2.

Exercise 2.12 ●● The constants in UCB1's bound

Repeat the proof of Theorem 2.6 for the index $\hat{\mu}_a + \sqrt{\alpha \ln(t-1)/N_a}$ with a constant $\alpha > 0$. (a) Show that Step 3 works with $\ell = \lceil 4\alpha \ln n/\Delta_a^2 \rceil$. (b) Show that the union bound of Step 4 gives a finite constant only if $\alpha > 3/2$, and find $\mathbb{E}[N_a(n)]$'s bound in terms of the Riemann zeta function $\zeta(p) = \sum_{m \geq 1} m^{-p}$. (c) Check that $\alpha = 2$ recovers the theorem, and discuss what smaller α buys and costs.

Exercise 2.13 ●● Tuning explore-then-commit

(a) Minimize the bound of Proposition 2.8 (replace $n - 2m$ by n) over real $m > 0$, and show that the minimum is $\frac{2}{\Delta}\big(1 + \ln \frac{n\Delta^2}{2}\big)$ when $n\Delta^2 > 2$. (b) Evaluate the tuned m and the bound for $\Delta = 0.2$ and $n = 10^4$. (c) Suppose Δ is unknown and you choose $m = \lceil n^{2/3} \rceil$. Show that the bound is $O(n^{2/3})$ uniformly over $\Delta \in (0, 1]$. (d) With `rlcourse.bandits.ExploreThenCommit`, measure the regret of the tuned version on the two-armed bandit with means 0.4 and 0.6, and compare with UCB1 and Thompson sampling.

Exercise 2.14 ●● The Lai–Robbins constant for Bernoulli arms

(a) Compute the Lai–Robbins constant $\Delta/\mathrm{kl}(\mu_a, \mu^*)$ for two-armed Bernoulli bandits with means $(0.4, 0.6)$, $(0.8, 0.9)$, $(0.1, 0.2)$ and $(0.01, 0.11)$, and compare each with UCB1's constant $8/\Delta$ and with $1/(2\Delta)$. (b) Use Pinsker's inequality $\mathrm{kl}(p, q) \geq 2(p-q)^2$ to show that the constant never exceeds $1/(2\Delta)$. (c) Explain in words why bandits with means near 0 or 1 are easier. (d) In Lab Task 4, add `rlcourse.bandits.KLUCB` and estimate the slopes of the regret curves against $\ln n$ between $n = 1000$ and $n = 5000$.

Exercise 2.15 ●● The LinUCB ellipsoid

(a) Show that $\max\{\theta^\top x : \|\theta - \hat{\theta}\|_V \leq \alpha\} = \hat{\theta}^\top x + \alpha\|x\|_{V^{-1}}$ for a positive definite V, and find the maximizer. (b) With $\lambda = 1$ and $d = 2$, the agent has pulled nine arms whose features were all $e_1 = (1, 0)$. Compute V and the bonuses of $x = e_1$ and $x = e_2$, and interpret. (c) Show that if the features of the K arms are the standard basis vectors $e_1, \ldots, e_K$ in every round, LinUCB becomes a UCB rule with estimate $\Sigma_a/(\lambda + N_a)$ and bonus $\alpha/\sqrt{\lambda + N_a}$, where Σ_a is the sum of arm a's rewards.

Exercise 2.16 ●●● When does Thompson sampling beat UCB?

In Figure 2.3 Thompson sampling's regret is less than a quarter of UCB1's. Is this a general fact? Discuss when Thompson sampling should be expected to beat UCB-type algorithms in practice and when it should not, considering at least: the reward noise relative to the worst case that UCB1's bonus assumes; prior knowledge; delayed and batched feedback; model misspecification and heavy tails; computational cost in large or structured problems; and the need for worst-case guarantees. Propose an experiment, runnable with `rlcourse.bandits`, that would test one of your claims, and cite the literature that supports your other claims.

Exercise 2.17 ●●● The value of information

A known arm pays λ in every round; an unknown Bernoulli arm has a Beta$(1, 1)$ prior. The agent maximizes $\mathbb{E}[\sum_{t \geq 1} \gamma^{t-1} r_t]$ with $\gamma = 0.9$. (a) Explain why, once it is optimal to play the known arm, it is optimal to play it forever. (b) Write the dynamic-programming equation for the value $W(S, F)$ of the problem when the unknown arm's posterior is Beta$(S + 1, F + 1)$. (c) Solve it numerically (truncate at $S + F = 600$) and find the largest λ for which the agent should try the unknown arm first: this is the Gittins index of the unknown arm. (d) Compare it with the arm's posterior mean and interpret the difference. What happens to the index as $\gamma \to 0$ and as $\gamma \to 1$?

CHAPTER 3

Markov Decision Processes

WHAT YOU WILL LEARN

- Explain the Markov property and decide whether a given description of a situation has it.
- Define a Markov decision process, a policy, the return and the value functions, and model a small decision problem in these terms.
- Derive the Bellman expectation equations and solve them for small problems, by hand and as a linear system.
- State the Bellman optimality equations and use them to find the optimal policy of the Trail for every discount factor.
- Prove that every finite discounted MDP has an optimal policy that is deterministic and stationary (●●).
- Define discounted occupancy measures and prove the performance difference lemma (●●).
- Describe the finite-horizon, average-reward and linear-programming formulations and say when each is used (●●●).

In the bandit problems of Chapter 2 the agent faced the same situation in every round. Its choice decided its reward, and nothing else: whatever it did, the next round looked exactly like the last. Most decisions are not like that. A move in chess changes the position in which the next move must be made. An order placed by a shop today decides how much stock it has tomorrow. On the Trail of Chapter 1, the action `go` earns nothing now but takes the hiker to a place where better rewards are available. Actions have consequences that outlast them, and a good decision must account for what it makes possible later.

The *Markov decision process* (MDP) is the mathematical model of such problems. It adds one ingredient to the bandit: a *state* that the agent's actions change. From this ingredient follow the central objects of reinforcement learning: policies, returns, value functions, and the *Bellman equations* that tie the value of a state to the values of the states that follow it. Almost every algorithm in this book, from dynamic programming to temporal-difference learning, policy gradients and the training of language models, is a way of solving or approximating these equations. This chapter also fixes the notation for the rest of the book.

We start with the Markov property (Section 3.1), define MDPs and write the Trail as one (Section 3.2), and introduce policies (Section 3.3), returns (Section 3.4) and value functions (Section 3.5). The heart of the chapter is the Bellman expectation equation (Section 3.6) and the Bellman optimality equation (Section 3.7), both worked out on the Trail and on a slippery version of it in which the hiker sometimes fails to move. Section 3.11 discusses what the MDP model leaves out and how to choose a model for a real problem. These sections form the Foundations

track. Three ●● and ●●● sections complete the theory: the proof that an optimal deterministic policy always exists (Section 3.8), occupancy measures and the performance difference lemma, which later chapters on policy optimization rely on (Section 3.9), and other objectives: finite horizons, average reward and linear programming (Section 3.10). On a first reading you can skip them; the ● sections state their main results in words.

3.1 From bandits to sequential decisions

Three examples show what changes when actions have lasting effects.

- Robot navigation. A delivery robot moves through a building. Each motion command changes its position, and its position decides which commands are useful next. A step that leads away from the goal may still be the right one if it goes around an obstacle.
- Inventory control. Each evening a shop decides how many units of a product to order. Ordering costs money now; running out of stock loses sales later; holding too much stock ties up money and space. Today's order is tomorrow's stock.
- The Trail. In Start the hiker can exit with +1 or go on with nothing. Going on is worth more only because of where it leads: to Middle, from which +2 is available at once and +4 one station later.

In each case two features are new compared with a bandit. First, there is a *state*: the robot's position, the shop's stock level, the hiker's station. The state carries the consequences of past actions into the future. Second, consequences are *delayed*: the worth of an action includes the rewards that it makes possible later, not only the reward it earns now. A good agent must therefore look ahead, and the rest of this chapter makes "looking ahead" precise.

3.1.1 The Markov property

What should the state contain? The guiding idea is that the state must summarize everything about the past that matters for the future. If we know the current state and the action, then knowing more about the past should not change our predictions about what happens next. In a phrase: *the future is independent of the past, given the present.*

Some examples make the idea concrete.

- The *position* of a thrown ball does not tell us where it will be a moment later; position and *velocity* together do. Position alone is not a Markov state; position and velocity are.
- A diagram of the pieces on a chess board is almost, but not quite, a Markov state. Whether castling is still allowed, whether an en-passant capture is possible, and how often the position has been repeated all depend on the history of the game and all affect what can happen next. Adding these few facts to the board makes the state Markov.
- The stock level of the shop is a Markov state if orders arrive overnight and the demand on each day is independent of the demand on earlier days. If orders take two days to arrive, the stock level is no longer enough: the state must also record what has been ordered but not yet delivered.

Formally, write $h_t = (s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_t)$ for the *history* up to time t. The state process has the *Markov property* if, for every time t, every history h_t, every action a, and all possible next states s' and rewards x,

$$\Pr(s_{t+1} = s', r_t = x \mid h_t, a_t = a) = \Pr(s_{t+1} = s', r_t = x \mid s_t, a_t = a). \tag{3.1}$$

In words: once the current state and action are known, the rest of the history carries no further information about the next state and reward.

> **PITFALL**
>
> The Markov property is a property of the *description* we choose, not of the world. Every process becomes Markov if the state is the entire history; but then no state is ever seen twice, and nothing learned in one situation helps in another. Choosing a state is a compromise between keeping enough information to predict the future and few enough distinct states to learn about each. When the agent cannot observe a Markov state at all, as a poker player cannot see the other players' cards, the problem is *partially observable*; Chapter 19 treats that case.

3.2 Markov decision processes

We can now define the model. Throughout the book we write $\mathcal{S}$ for the set of states, $\mathcal{A}$ for the set of actions, and use the *reward convention* of Chapter 1: r_t is the reward received right after the action a_t is taken in the state s_t, so that a single step of interaction is the *transition* (s_t, a_t, r_t, s_{t+1}).

Definition 3.1 Markov decision process

A *finite Markov decision process* is a tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma, d_0)$ in which

- $\mathcal{S}$ is a finite set of *states* and $\mathcal{A}$ a finite set of *actions*;
- $P(s' \mid s, a)$ is the *transition probability* of moving to state s' when action a is taken in state s, with $\sum_{s'} P(s' \mid s, a) = 1$;
- $r(s, a)$ is the *expected reward* for taking action a in state s;
- $\gamma \in [0, 1)$ is the *discount factor*;
- d_0 is the *initial-state distribution*.

The interaction starts in $s_0 \sim d_0$. At each time $t = 0, 1, 2, \dots$ the agent chooses an action a_t; the environment returns a reward r_t and a next state s_{t+1} such that, whatever the history h_t,

$$\Pr(s_{t+1} = s' \mid h_t, a_t) = P(s' \mid s_t, a_t), \qquad \mathbb{E}[r_t \mid h_t, a_t] = r(s_t, a_t).$$

We assume that rewards are bounded: $|r_t| \le r_{\max}$ for some constant $r_{\max}$.

A few remarks on this definition.

Rewards. The reward may be random: a slot machine pays out sometimes and not at other times. The MDP records only its expected value $r(s, a)$, because expected returns, and therefore values and optimal policies, depend only on expected rewards. (The spread of the rewards matters for

how hard values are to estimate from samples, Chapters 5 and 6, and for distributional methods, Chapter 10.) Some books let the reward depend on the next state as well, $r(s, a, s')$; this fits our definition with $r(s, a) = \sum_{s'} P(s' \mid s, a)\, r(s, a, s')$. Sutton and Barto write the reward that follows a_t as R_{t+1} (Sutton and Barto 2018); it is the same quantity as our r_t with a different index.

Actions that depend on the state. In many problems some actions are available only in some states, as on the Trail, where Goal offers only finish. We then write $\mathcal{A}(s)$ for the actions available in s. To keep formulas short we usually write $\mathcal{A}$ and let the reader restrict maxima and sums to $\mathcal{A}(s)$.

Episodic and continuing tasks. A task is *episodic* if it breaks into episodes that end in a terminal state, like a game or a trip along the Trail. It is *continuing* if it goes on without end, like the control of a power grid. A single formalism covers both: we model a terminal state as an *absorbing* state, one that the process never leaves and in which every action earns reward 0. After an episode ends, the process stays in the terminal state forever and collects nothing more, so the total discounted reward is the same as if it had stopped. The code in `rlcourse` follows this convention: the Trail has a fourth state, End, with $P(\text{End} \mid \text{End}, a) = 1$ and $r(\text{End}, a) = 0$.

The discount factor. Section 3.4 explains the role of γ. Episodic tasks in which every policy reaches a terminal state with probability 1 also make sense with $\gamma = 1$; the Trail is one. With $\gamma = 1$ some of the arguments below need extra care, and we point this out where it matters.

3.2.1 The Trail as an MDP

The Trail of Section 1.3 is a finite MDP with four states and two actions. Table 3.1 lists its transitions; the only random element is the policy, because every action has a single outcome.

Table 3.1. The Trail as an MDP. Episodes start in Start, so $d_0(\text{Start}) = 1$, and End is absorbing.

State s	Action a	Next state s'	$P(s' \mid s, a)$	$r(s, a)$
Start	go	Middle	1	0
Start	exit	End	1	1
Middle	go	Goal	1	0
Middle	exit	End	1	2
Goal	finish	End	1	4
End	any	End	1	0

The plain Trail is a little too simple to show what the theory of this chapter is for: every episode lasts at most three steps, so we could compute everything by listing paths, as Chapter 1 did. From now on we also use a variant with genuine uncertainty. On the *slippery Trail* (Figure 3.1) each go succeeds only with probability 0.8; with probability 0.2 the hiker slips and stays where she is, earning nothing. Exits and finish are unchanged. So

$$P(\text{Middle} \mid \text{Start}, \text{go}) = 0.8, \qquad P(\text{Start} \mid \text{Start}, \text{go}) = 0.2,$$

and similarly in Middle. The slippery Trail has loops: a hiker who keeps choosing go can slip any number of times, so there are infinitely many possible episodes, and listing them is no longer an option. We need a better tool, and Section 3.6 provides it.

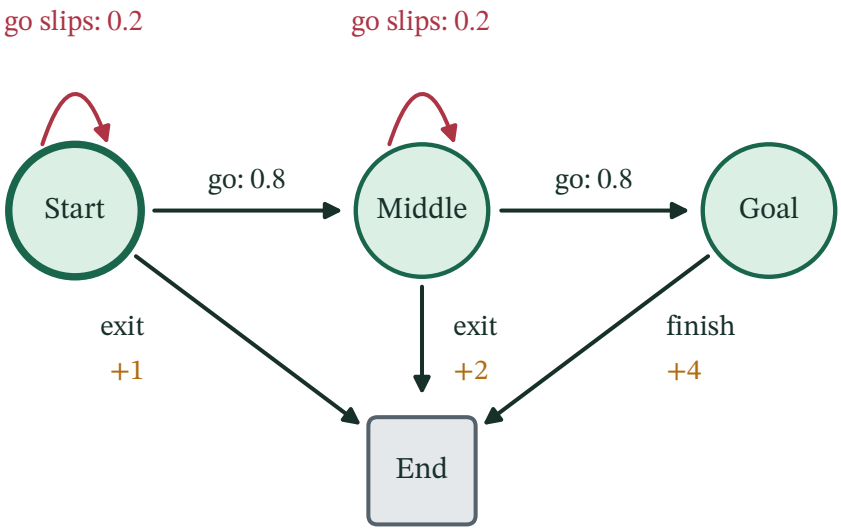


Figure 3.1. The slippery Trail. Each go moves the hiker on with probability 0.8 and leaves her where she is with probability 0.2 (the loops). Exits and finish work as on the Trail. Because of the loops, an episode can last arbitrarily long.

In code, `Trail(slip=0.2).to_mdp(gamma)` returns the slippery Trail as a `FiniteMDP` with arrays `P[s, a, s2]` and `R[s, a]`; states are numbered 0 to 3 (Start, Middle, Goal, End) and actions 0 (exit) and 1 (go). In Goal both action indices mean finish.

Example 3.1 A small inventory problem

A shop sells one product and can store at most two units. Each evening it sees its stock $s \in \{0, 1, 2\}$ and orders a units, with $s + a \le 2$; the order arrives overnight. The next day's demand D is 0, 1 or 2 units with probabilities 0.3, 0.5 and 0.2, independently of the past. The shop sells $\min(s + a, D)$ units at a price of 4 each and pays 1 for each unit it orders. The next evening's stock is $s' = s + a - \min(s + a, D)$.

This is an MDP with three states. In state s the available actions are $\mathcal{A}(s) = \{0, \dots, 2 - s\}$, so there are $3 + 2 + 1 = 6$ state–action pairs. Take $s = 0$ and $a = 1$: the shop starts the day with one unit. If nobody comes ($D = 0$, probability 0.3) the unit is still there in the evening; otherwise it is sold. So

$$P(1 \mid 0, 1) = 0.3, \qquad P(0 \mid 0, 1) = 0.7, \qquad r(0, 1) = 4 \cdot 0.7 - 1 = 1.8.$$

The state is Markov because demand on different days is independent and orders arrive overnight. With a two-day delivery delay, the state would also need to contain the order placed the evening before (Exercise 3.6 builds the full model).

♦

3.3 Policies

A policy is the agent's rule for choosing actions. The most general rule may look at the entire history and may randomize; the simplest looks only at the current state and picks one action. Both extremes, and the cases in between, appear in this book, so we name them.

Definition 3.2 Policy

A *policy* is a rule for choosing actions. In general it is *history-dependent* and *randomized*: at each time t it gives a probability distribution $\pi_t(\cdot \mid h_t)$ over actions that may depend on the whole history h_t. A policy is *Markov* if its choice depends only on the time t and the current state s_t, and *stationary* if it is Markov and the same at every time step; a stationary policy is described by the probabilities $\pi(a \mid s)$ of taking action a in state s. A policy is *deterministic* if it puts probability 1 on a single action; a deterministic stationary policy is a function $\pi(s)$ from states to actions. Unless we say otherwise, "policy" means a stationary policy $\pi(a \mid s)$.

The Trail's policies are described by $p_0 = \pi(\mathsf{go} \mid \mathsf{Start})$ and $p_1 = \pi(\mathsf{go} \mid \mathsf{Middle})$, as in Chapter 1. A finite MDP has $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic stationary policies, one choice per state; the Trail has $2 \times 2 = 4$ that behave differently, because only the choices in Start and Middle matter.

A stationary policy and the MDP together determine the probability of every finite trajectory. By the Markov property, the probability that the first n steps visit the states $s_0, \dots, s_n$ with actions $a_0, \dots, a_{n-1}$ is

$$\Pr_{\pi}(s_0, a_0, s_1, a_1, \dots, s_n) = d_0(s_0) \prod_{t=0}^{n-1} \pi(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t). \tag{3.2}$$

The policy enters through the factors $\pi(a_t \mid s_t)$ only. This simple fact is the starting point of the policy gradient methods of Chapter 11. We write $\Pr_\pi$ and $\mathbb{E}_\pi$ for probabilities and expectations of trajectories generated in this way.

Why allow history-dependent policies at all, if we mostly use stationary ones? Because we want to know whether looking at the history could ever help. For fully observed MDPs with discounting, it cannot: Section 3.8 proves that some deterministic stationary policy is at least as good as every other policy, history-dependent or randomized, in every state. History matters when the state is not fully observed (Chapter 19), and randomization matters in games against adversaries (Chapter 21) and under constraints (Section 3.10).

3.4 Returns and discounting

The agent's goal concerns the whole sequence of rewards, not any single one. We measure a sequence of rewards by its discounted sum.

Definition 3.3 Return

The *return* from time t is the discounted sum of the rewards that follow,

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{3.3}$$

In an episodic task that ends at time T the sum stops at r_{T-1}; with terminal states modeled as absorbing, zero-reward states, the infinite sum gives the same number.

Because $|r_t| \le r_{\max}$ and $\gamma < 1$, the sum converges and is bounded by a geometric series:

$$|G_t| \le \sum_{k=0}^{\infty} \gamma^k r_{\max} = \frac{r_{\max}}{1-\gamma}.$$

Splitting off the first term of (3.3) gives the *recursive identity* of returns,

$$G_t = r_t + \gamma\big(r_{t+1} + \gamma r_{t+2} + \cdots\big) = r_t + \gamma\, G_{t+1}. \tag{3.4}$$

In words: the return from now is the reward now plus the discounted return from the next step. This one-line identity is the seed of every Bellman equation in the book.

Why discount? There are several reasons, and in a given problem one or more of them may apply.

1. *Finite sums.* In a continuing task the undiscounted sum of rewards is usually infinite or undefined, and infinite returns cannot be compared. Discounting makes every return finite.
2. *Uncertainty about the future.* If at every step there is a probability $1-\gamma$ that the process stops for reasons outside the model (the battery dies, the customer leaves, the market closes), then a reward k steps away is collected only with probability γ^k. The discounted return is then the expected undiscounted return.
3. *Preference for sooner rewards.* People and firms prefer a reward now to the same reward later. In economics, an interest rate ρ per period corresponds to $\gamma = 1/(1+\rho)$: a dollar next year is worth γ dollars today.
4. *A modeling and learning choice.* Smaller γ makes the problem easier to solve and to learn, because it makes the far future matter less. Practitioners often choose γ smaller than the problem would suggest, trading a slightly different objective for faster and more stable learning (see Chapter 23).

The effective horizon. The weights $1, \gamma, \gamma^2, \ldots$ add up to $1/(1-\gamma)$, and a reward $1/(1-\gamma)$ steps away has weight about $e^{-1} \approx 0.37$. So $1/(1-\gamma)$ is a useful measure of how far ahead the agent looks, the *effective horizon* (Table 3.2). It is not a hard cutoff: rewards beyond it still count, only less.

Table 3.2. Discount factors and how far ahead they look.

γ	effective horizon $1/(1-\gamma)$	weight γ^{10} of a reward 10 steps away	weight γ^{100}
0.9	10	0.349	0.00003
0.99	100	0.904	0.366
0.999	1000	0.990	0.905

On the Trail, as Chapter 1 showed, the discount factor changes which behavior is best: going all the way pays only when $\gamma > 0.5$. The discount factor is part of the problem, and choosing it is a modeling decision (Section 3.11).

The reward hypothesis. Everything here rests on the assumption that the goal can be expressed by a single scalar reward whose expected discounted sum is to be maximized. Sutton and Barto (2018, sec. 3.2) call this the *reward hypothesis*, and Silver et al. (2021) argue that reward maximization may be enough to explain intelligent behavior in general. Chapter 1 discussed the debate. For this chapter two practical points matter. First, the reward should say *what* we want achieved, not *how*: a reward for intermediate steps that we believe are helpful can be exploited in ways we did not intend. Second, some goals are not naturally a sum of rewards: constraints ("never exceed this temperature"), risk ("avoid a large loss even if the average is good") or several objectives that cannot be traded at a fixed rate. Constrained MDPs (Section 3.10 and Chapter 23) and distributional methods (Chapter 10) address some of these; reward design is the subject of Chapter 23, and learning rewards from people is the subject of Chapter 18.

PITFALL

Adding the same constant to every reward does not change which behavior is best in a continuing task, but it can in an episodic one. A negative constant ("a cost per step") makes ending the episode early attractive; a positive one makes prolonging it attractive. The reason is that the terminal state, with its zero reward, is not shifted. Exercise 3.8 works this out on the Trail.

3.5 Value functions

A policy produces random returns. To compare policies, and to decide what to do in a state, we need their *expected* returns. These are the value functions, the central objects of reinforcement learning.

Definition 3.4 Value functions

For a stationary policy π, the *state-value function* V^π, the *action-value function* Q^π and the *advantage function* A^π are

$$V^\pi(s) = \mathbb{E}_\pi\big[G_0 \mid s_0 = s\big],$$
$$Q^\pi(s,a) = \mathbb{E}_\pi\big[G_0 \mid s_0 = s,\ a_0 = a\big],$$
$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

In $Q^\pi(s, a)$ the first action is fixed to a, and π is followed from the next step on.

$V^\pi(s)$ answers the question "how good is it to be in s, if I follow π?". $Q^\pi(s, a)$ answers "how good is it to take a in s and follow π afterward?", and $A^\pi(s, a)$ says how much better or worse a is than what π does on average in s. Because neither the MDP nor a stationary policy changes over time, the expected return from a state is the same whenever the state is reached: $V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$ for every time t at which s can occur. The notation table writes the values this way.

The two value functions are tied together by two relations. The first averages over the policy's choice of action:

$$V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s, a). \tag{3.5}$$

The second averages over the environment's response. Taking a in s earns $r(s, a)$ on average and leads to s' with probability $P(s' \mid s, a)$; from there on, by the Markov property, the future depends on the past only through s', so its expected discounted return is $V^\pi(s')$:

$$Q^\pi(s, a) = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V^\pi(s'). \tag{3.6}$$

A consequence of (3.5) is that the advantage averages to zero under its own policy: $\sum_a \pi(a \mid s) A^\pi(s, a) = 0$ in every state. Some actions are better than the policy's average and some are worse.

On the Trail with the uniform policy and $\gamma = 0.9$, Chapter 1 found $V^\pi(\text{Middle}) = 2.8$ and $V^\pi(\text{Start}) = 1.76$. Then (3.6) gives $Q^\pi(\text{Start}, \text{go}) = 0 + 0.9 \times 2.8 = 2.52$ and $Q^\pi(\text{Start}, \text{exit}) = 1$, and (3.5) recovers $V^\pi(\text{Start}) = \frac{1}{2} \times 2.52 + \frac{1}{2} \times 1 = 1.76$. The advantages in Start are $A^\pi(\text{Start}, \text{go}) = 0.76$ and $A^\pi(\text{Start}, \text{exit}) = -0.76$: going on is better than the coin flip, and exiting is worse.

3.6 The Bellman expectation equations

The relations (3.5) and (3.6) each express one value function in terms of the other. Substituting one into the other gives an equation for each value function alone. These are the *Bellman equations*, named after Richard Bellman, who made them the basis of dynamic programming (Bellman 1957b).

In words. The value of a state is the expected immediate reward plus the discounted expected value of the state that comes next. Nothing more is needed, because the value of the next state already accounts for everything that happens after it.

Formally. Start from the definition of V^π and the recursive identity (3.4), then condition on the first action and the next state:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\big[r_0 + \gamma G_1 \mid s_0 = s\big] \\ &= \sum_a \pi(a \mid s)\Big[r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, \underbrace{\mathbb{E}_\pi\big[G_1 \mid s_0 = s,\, a_0 = a,\, s_1 = s'\big]}_{= V^\pi(s') \text{ by the Markov property}}\Big]. \end{aligned}$$

Given $s_1 = s'$, the rewards from time 1 on are produced by the same policy and the same transition probabilities as if the process had started in s', whatever s_0 and a_0 were; so their expected discounted sum is $V^\pi(s')$. This gives the *Bellman expectation equation* for V^π:

$$V^\pi(s) = \sum_a \pi(a \mid s)\Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^\pi(s')\Big] \quad \text{for every } s \in \mathcal{S}. \tag{3.7}$$

Substituting (3.5) into (3.6) in the same way gives the equation for Q^π:

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \sum_{a'} \pi(a' \mid s')\, Q^\pi(s',a') \quad \text{for every } s,a. \tag{3.8}$$

In words: the value of a state–action pair is its expected reward plus the discounted value of the next pair, where the next state is chosen by the environment and the next action by the policy.

Equation (3.7) is a system of $|\mathcal{S}|$ linear equations in the $|\mathcal{S}|$ unknowns $V^\pi(s)$, one equation per state. It turns the evaluation of a policy, which by definition involves the infinitely many possible futures, into a finite problem.

Backup diagrams (Figure 3.2) are a compact way to picture such equations. Each diagram shows how the value at its root is computed from the values one step later: open circles are states, filled dots are state–action pairs, and the value of the root is "backed up" from the leaves. The diagrams for V^* and Q^* belong to Section 3.7.

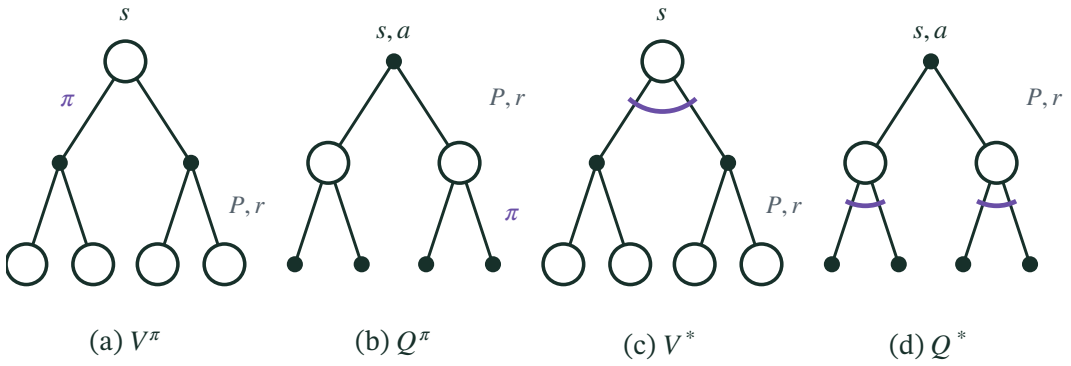


Figure 3.2. Backup diagrams for (a) V^π, (b) Q^π, (c) V^* and (d) Q^*. Open circles are states and filled dots are actions. In (a) the policy π chooses an action and the environment (P, r) chooses the next state; the value at the root is an average over both choices. In (c) and (d) the arc marks a maximum over actions instead of an average.

Example 3.2 The Trail's uniform policy, by the Bellman equation

Take the uniform policy ($p_0 = p_1 = \frac{1}{2}$) and $\gamma = 0.9$. The terminal state has $V^\pi(\text{End}) = 0$. In Goal the only action earns 4 and leads to End:

$$V^\pi(\text{Goal}) = 4 + 0.9 \times 0 = 4.$$

In Middle, half the time the hiker exits (2, then nothing) and half the time she goes on (0, then Goal):

$$V^\pi(\text{Middle}) = \tfrac{1}{2}\big[2 + 0.9 \times 0\big] + \tfrac{1}{2}\big[0 + 0.9 \times 4\big] = 1 + 1.8 = 2.8.$$

In Start:

$$V^\pi(\text{Start}) = \tfrac{1}{2}\big[1 + 0.9 \times 0\big] + \tfrac{1}{2}\big[0 + 0.9 \times 2.8\big] = 0.5 + 1.26 = 1.76.$$

These are the values Chapter 1 found by listing all paths. The difference is in how the work is organized. Listing paths handles each complete episode separately; the Bellman equation handles each *state* once and reuses its value: $V^\pi(\text{Middle})$ is computed once and then used by Start. In a problem whose episodes last n steps the number of paths can grow exponentially with n, while the number of equations stays equal to the number of states. For a general policy the same steps give

$$V^\pi(\text{Middle}) = 2(1 - p_1) + 4\gamma p_1, \qquad V^\pi(\text{Start}) = (1 - p_0) + \gamma p_0\, V^\pi(\text{Middle}),$$

which, expanded, is the formula for $J(p_0, p_1)$ in Chapter 1.

♦

3.6.1 The matrix form

For a fixed policy, the Bellman equation is linear, and it is convenient to write it with vectors and matrices. Number the states $1, \dots, |\mathcal{S}|$ and treat a value function V as a vector in $\mathbb{R}^{|\mathcal{S}|}$. The policy and the MDP together define a *policy transition matrix* P^π and an *expected reward vector* r^π:

$$P^\pi(s, s') = \sum_a \pi(a \mid s)\, P(s' \mid s, a), \qquad r^\pi(s) = \sum_a \pi(a \mid s)\, r(s, a).$$

$P^\pi(s, s')$ is the probability of moving from s to s' in one step when following π; each row of P^π is a probability distribution, so P^π is a *stochastic matrix*. With this notation, (3.7) reads

$$V^\pi = r^\pi + \gamma P^\pi V^\pi, \qquad \text{so} \qquad V^\pi = (I - \gamma P^\pi)^{-1} r^\pi, \tag{3.9}$$

where I is the identity matrix. The second form requires the matrix $I - \gamma P^\pi$ to be invertible. It always is.

Proposition 3.5 The Bellman expectation equation has a unique solution

Let $\gamma \in [0, 1)$ and let π be a stationary policy. Then $I - \gamma P^\pi$ is invertible,

$$(I - \gamma P^\pi)^{-1} = \sum_{k=0}^{\infty} \gamma^k (P^\pi)^k,$$

its entries are nonnegative, and each of its rows sums to $1/(1 - \gamma)$. In particular, V^π is the only solution of the Bellman expectation equation (3.7).

Proof. For any vector x, each entry of $P^\pi x$ is an average of entries of x, so $\max_s |(P^\pi x)(s)| \le \max_s |x(s)|$. If $(I - \gamma P^\pi)x = 0$, then $x = \gamma P^\pi x$, so $\max_s |x(s)| \le \gamma \max_s |x(s)|$, which for $\gamma < 1$ forces $x = 0$. Hence $I - \gamma P^\pi$ has a trivial null space and is invertible. The partial sums $S_n = \sum_{k=0}^{n-1} \gamma^k (P^\pi)^k$ satisfy $(I - \gamma P^\pi) S_n = I - \gamma^n (P^\pi)^n$, and the entries of $(P^\pi)^n$ lie in $[0, 1]$, so $\gamma^n (P^\pi)^n \to 0$ and $S_n \to (I - \gamma P^\pi)^{-1}$. Every power $(P^\pi)^k$ is a stochastic matrix, with nonnegative entries and rows summing to 1; so the limit has nonnegative entries and row sums $\sum_k \gamma^k = 1/(1-\gamma)$. Finally, the derivation above shows that V^π satisfies (3.7), and the equation has only one solution. ■

The series has a direct meaning. The entry $(P^\pi)^k(s, s')$ is the probability of being in s' after k steps from s, so $\big((P^\pi)^k r^\pi\big)(s) = \mathbb{E}_\pi[r_k \mid s_0 = s]$ is the expected reward k steps ahead, and

$$V^\pi = \sum_{k=0}^{\infty} \gamma^k (P^\pi)^k r^\pi$$

is the definition of V^π term by term: the expected discounted sum of future rewards.

For the Trail's uniform policy, with the states in the order Start, Middle, Goal, End,

$$P^\pi = \begin{pmatrix} 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad r^\pi = \begin{pmatrix} 0.5 \\ 1 \\ 4 \\ 0 \end{pmatrix},$$

and solving $(I - 0.9P^\pi)V = r^\pi$ gives $V^\pi = (1.76, 2.8, 4, 0)$ again.

> **PITFALL**
>
> In code, solve the linear system (`np.linalg.solve`) rather than forming the inverse matrix; it is faster and more accurate. Either way the cost grows like $|\mathcal{S}|^3$, which is fine for thousands of states and hopeless for millions; Chapter 4 gives iterative methods whose cost per step grows like the number of nonzero transition probabilities. Note also that with $\gamma = 1$ the matrix $I - P^\pi$ is always singular, because each of its rows sums to zero; for episodic problems with $\gamma = 1$, solve only for the non-terminal states, whose values are what you want anyway (Exercise 3.9).

3.6.2 The slippery Trail: a coupled system

On the plain Trail, the Bellman equations can be solved one at a time from the end, because no state can be revisited. The slippery Trail is different, and it shows why we need the linear-algebra view.

Example 3.3 The slippery Trail's uniform policy

Take the uniform policy on the slippery Trail with $\gamma = 0.9$. Still $V^\pi(\text{Goal}) = 4$. In Middle, an exit earns 2; a go reaches Goal with probability 0.8 and stays in Middle with probability 0.2:

$$V^\pi(\text{Middle}) = \tfrac{1}{2} \times 2 + \tfrac{1}{2} \times 0.9\big[0.8 \times 4 + 0.2\, V^\pi(\text{Middle})\big] = 2.44 + 0.09\, V^\pi(\text{Middle}).$$

The unknown appears on both sides, because the hiker can come back to Middle. Solving, $0.91\, V^\pi(\text{Middle}) = 2.44$, so $V^\pi(\text{Middle}) = 244/91 \approx 2.681$. In Start,

$$\begin{aligned} V^\pi(\text{Start}) &= \tfrac{1}{2} \times 1 + \tfrac{1}{2} \times 0.9\big[0.8\, V^\pi(\text{Middle}) + 0.2\, V^\pi(\text{Start})\big] \\ &= 0.5 + 0.36\, V^\pi(\text{Middle}) + 0.09\, V^\pi(\text{Start}), \end{aligned}$$

so $0.91\, V^\pi(\text{Start}) = 0.5 + 0.36 \times 2.681 = 1.465$ and $V^\pi(\text{Start}) \approx 1.610$. Slipping costs the uniform policy about 0.12 in Middle and 0.15 in Start compared with the plain Trail. The action values

follow from (3.6): $Q^\pi(\text{Start}, \text{go}) = 0.9\,(0.8 \times 2.681 + 0.2 \times 1.610) \approx 2.220$ and $Q^\pi(\text{Middle}, \text{go}) = 0.9\,(0.8 \times 4 + 0.2 \times 2.681) \approx 3.363$.

In matrix form, the rows of P^π for Start and Middle are $(0.1, 0.4, 0, 0.5)$ and $(0, 0.1, 0.4, 0.5)$: the diagonal entries $0.1 = \frac{1}{2} \times 0.2$ are the loops. Here we could still solve the equations one by one, from Middle back to Start; in an MDP where states can be revisited along longer cycles, all the equations are coupled and must be solved together, as one linear system.



3.7 Optimality

So far we have evaluated given policies. The goal, though, is to find a good one. What does "best" mean, when a policy has a value in every state?

We say that π is *at least as good as* π' if $V^\pi(s) \geq V^{\pi'}(s)$ in every state s. This ordering is *partial*: two policies can each be better in some state. On the Trail with $\gamma = 0.9$, the policy that exits in Start but goes on in Middle has values 1 and 3.6 there, while the policy that goes on in Start and exits in Middle has 1.8 and 2. Neither is at least as good as the other. It is therefore a remarkable fact, proved in Section 3.8, that every finite MDP has a policy that is at least as good as *every* other policy, in *every* state at once.

Definition 3.6 Optimal values and optimal policies

The *optimal state-value function* is

$$V^*(s) = \sup_\pi V^\pi(s) \quad \text{for every } s \in \mathcal{S},$$

where the supremum is over all policies, including history-dependent and randomized ones. The *optimal action-value function*

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V^*(s')$$

is the expected return of taking a in s and behaving optimally afterward. A policy π^* is *optimal* if $V^{\pi^*}(s) = V^*(s)$ for every state s.

The Bellman optimality equations. Suppose we already know how to behave optimally from the next step on. Then the best thing to do now is to choose the action with the largest immediate reward plus discounted optimal value of the next state, and the value of doing so is $V^*(s)$. This reasoning, which Section 3.8 turns into a proof, gives the *Bellman optimality equation*:

$$V^*(s) = \max_a \Big[r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V^*(s') \Big] \quad \text{for every } s \in \mathcal{S}. \tag{3.10}$$

In words: the optimal value of a state is the value of its best action, when every later decision is also made optimally. Since the bracket is $Q^*(s, a)$, the equation says $V^*(s) = \max_a Q^*(s, a)$, and substituting this into the definition of Q^* gives the optimality equation for action values:

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q^*(s', a') \quad \text{for every } s, a. \tag{3.11}$$

Compared with the expectation equations (3.7) and (3.8), the average over the policy's actions has been replaced by a maximum; the backup diagrams (c) and (d) of Figure 3.2 show the maximum as an arc. The maximum makes these equations nonlinear, so they cannot be solved by a single linear solve. Chapter 4 shows how to solve them.

Greedy policies. A deterministic policy π is *greedy* with respect to a value function V if in every state it picks an action that looks best one step ahead:

$$\pi(s) \in \arg\max_a \Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s') \Big],$$

and greedy with respect to an action-value function Q if $\pi(s) \in \arg\max_a Q(s,a)$. The main facts about optimality, proved in Section 3.8 as Theorem 3.7, are these:

1. V^* is the *only* function that satisfies the optimality equation (3.10).
2. Every policy that is greedy with respect to V^* (equivalently, with respect to Q^*) is optimal. In particular, an optimal policy exists that is deterministic and stationary, and it is optimal in every state against every other policy, including history-dependent and randomized ones.
3. A stationary policy is optimal if and only if its own value function satisfies the optimality equation. So to check a candidate policy, compute its values and check that in no state does any single action do better than the policy's own choice.

Optimal policies need not be unique, because several actions may tie for the maximum, but the optimal value function is unique. A stochastic policy is optimal exactly when it only uses actions that maximize Q^*.

> **KEY IDEA**
>
> The optimality equation is a *local* condition: it says that no change of a single action, followed by optimal behavior, can do better. The theorem says that this local condition is enough. A policy whose values cannot be improved by any one-step deviation is optimal globally, against all policies.

Example 3.4 The Trail's optimal policy for every discount factor

Solve the optimality equations of the Trail backward from the end. $V^*(\text{End}) = 0$ and $V^*(\text{Goal}) = 4$. In Middle the hiker compares exiting with going on to Goal:

$$Q^*(\text{Middle}, \text{exit}) = 2, \qquad Q^*(\text{Middle}, \text{go}) = 4\gamma, \qquad V^*(\text{Middle}) = \max\{2, 4\gamma\}.$$

In Start:

$$Q^*(\text{Start}, \text{exit}) = 1, \qquad Q^*(\text{Start}, \text{go}) = \gamma\, V^*(\text{Middle}), \qquad V^*(\text{Start}) = \max\{1, \gamma V^*(\text{Middle})\}.$$

With $\gamma = 0.9$: $V^*(\text{Middle}) = \max\{2, 3.6\} = 3.6$ and $V^*(\text{Start}) = \max\{1, 3.24\} = 3.24$, so the optimal policy goes all the way. Figure 3.3 compares these values with those of the uniform policy. Note that $Q^*(\text{Start}, \text{go}) = 3.24$ is larger than $Q^\pi(\text{Start}, \text{go}) = 2.52$ for the uniform policy: the value of an action depends on what is done afterward.

Figure 3.4 shows the four optimal action values as functions of γ. For $\gamma < 0.5$, exiting is better in both states, $V^*(\text{Middle}) = 2$ and $V^*(\text{Start}) = 1$; for $\gamma > 0.5$ going on is better in both; at $\gamma = 0.5$ the two actions tie in both states and every policy is optimal.

♦

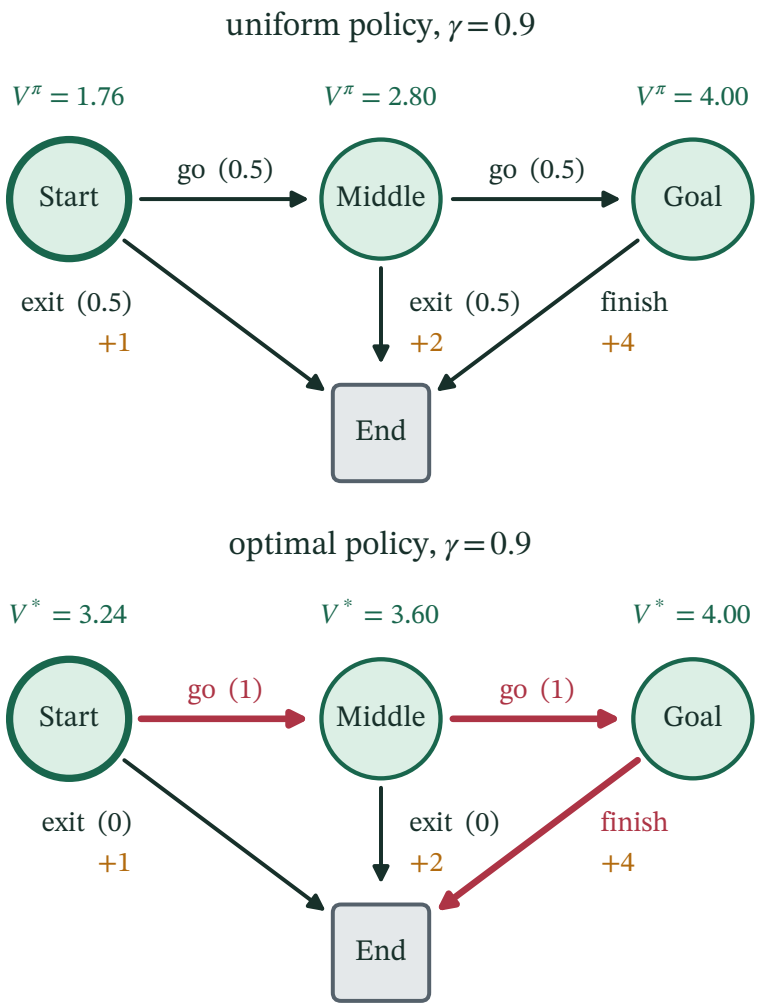


Figure 3.3. The Trail's values with $\gamma = 0.9$. Top: the uniform policy, with the probability of each action in parentheses. Bottom: the optimal policy, which goes all the way (highlighted). The optimal values are larger in every state, as they must be.

Guess and verify on the slippery Trail. Point 3 above gives a practical method for small problems: guess a policy, compute its values with the expectation equation, and check the optimality equation. On the slippery Trail with $\gamma = 0.9$, guess "always go". Its values satisfy

$$V(\text{Middle}) = 0.9\big[0.8 \times 4 + 0.2\,V(\text{Middle})\big], \qquad V(\text{Start}) = 0.9\big[0.8\,V(\text{Middle}) + 0.2\,V(\text{Start})\big],$$

which give $V(\text{Middle}) = 2.88/0.82 \approx 3.512$ and $V(\text{Start}) = 0.72 \times 3.512/0.82 \approx 3.084$. Now check: exiting would give $2 < 3.512$ in Middle and $1 < 3.084$ in Start. No one-step deviation helps, so "always go" is optimal, with $V^*(\text{Middle}) \approx 3.512$ and $V^*(\text{Start}) \approx 3.084$.

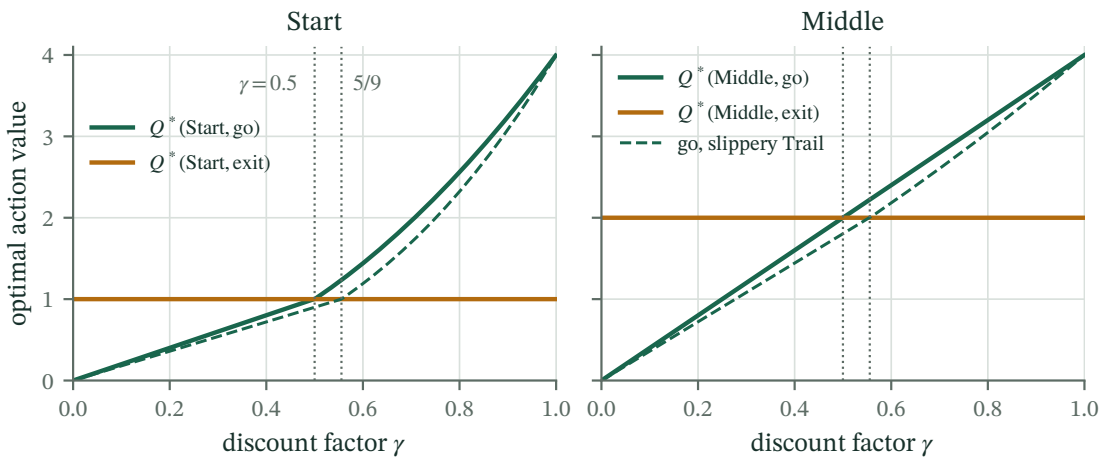


Figure 3.4. Optimal action values of the Trail in Start (left) and Middle (right) as functions of the discount factor. In both states the go and exit curves cross at $\gamma = 0.5$: below it the hiker should exit, above it she should go on. The dashed curves are the go values of the slippery Trail, which cross the exits later, at $\gamma = 5/9$.

There is a neat way to see the effect of slipping. Starting a go in Middle, the hiker reaches Goal after n attempts with probability $0.2^{n-1} \times 0.8$, so the expected discount until she arrives is

$$\tilde{\gamma} = \sum_{n=1}^{\infty} 0.2^{n-1} \times 0.8 \times \gamma^n = \frac{0.8\,\gamma}{1 - 0.2\,\gamma}.$$

The slippery Trail with "always go" behaves like the plain Trail with discount $\tilde{\gamma}$ per station; with $\gamma = 0.9$, $\tilde{\gamma} = 0.72/0.82 \approx 0.878$, and indeed $V^*(\text{Middle}) = 4\tilde{\gamma} \approx 3.512$ and $V^*(\text{Start}) = 4\tilde{\gamma}^2 \approx 3.084$. Going on is optimal when $\tilde{\gamma} > 0.5$, that is, when $\gamma > 5/9 \approx 0.556$: slipping makes the hiker effectively less patient.

A grid world. Figure 3.5 shows the optimal values and an optimal policy of a classic example, the 4×3 world of Russell and Norvig (2021). The agent starts in the bottom-left cell and moves up, down, left or right; a move goes in the intended direction with probability 0.8 and slips to each perpendicular direction with probability 0.1; moving into the wall (dark cell) or the border leaves the agent in place. Every move costs 0.04 (reward -0.04), and the cells marked $+1$ and -1 end the episode with those rewards. With $\gamma = 0.9$, an agent in the middle of the bottom row heads right and then up the third column, past the -1 cell; only from the start cell does it go up the left side. Russell and Norvig use $\gamma = 1$ (right panel); then the agent is more patient and, from the bottom row, takes the long safe way around the left. (With $\gamma = 1$ the contraction arguments of Chapter 4 do not apply directly, but because every step costs something, a policy that never reaches an exit has value $-\infty$, and dynamic programming still converges to the values shown (Bertsekas and Tsitsiklis 1996, ch. 2).)

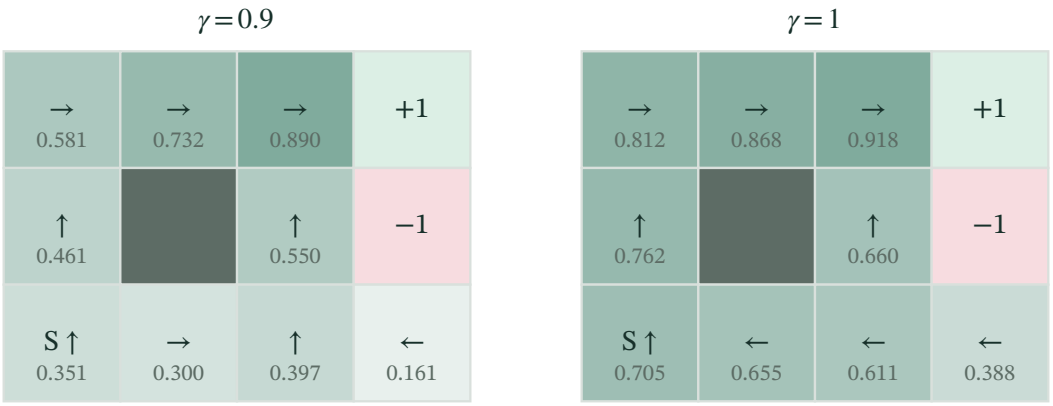


Figure 3.5. Optimal values (small numbers) and an optimal policy (arrows) of the 4×3 world. Left: $\gamma = 0.9$. Right: $\gamma = 1$, the values of Russell and Norvig. The more patient agent on the right avoids the -1 cell by going around the left side; the discounting agent on the left takes the shorter path from the middle of the bottom row.

CHECKPOINT

1. On the Trail with $\gamma = 0.9$, what is $Q^*(\text{Middle}, \text{exit})$, and why is it smaller than $V^*(\text{Middle})$?
2. With $\gamma = 0.4$, what is $V^*(\text{Start})$, and which policy achieves it?
3. Why is the uniform policy not optimal when $\gamma = 0.9$, even though it sometimes goes all the way?

Answers.

1. $Q^*(\text{Middle}, \text{exit}) = 2$: exiting earns 2 and ends the episode. It is smaller than $V^*(\text{Middle}) = 3.6$ because the optimal action in Middle is go.
2. $V^*(\text{Start}) = 1$, by exiting at once ($Q^*(\text{Start}, \text{go}) = 0.4 \times 2 = 0.8 < 1$).
3. A stochastic policy is optimal only if every action it uses maximizes Q^*. The uniform policy uses exit in Start, where $Q^*(\text{Start}, \text{exit}) = 1 < 3.24 = Q^*(\text{Start}, \text{go})$.

LAB

Lab 3.1: MDPs in code (`labs/ch03_mdp.py`, about 40 minutes). Run `python3 labs/ch03_mdp.py`. You will build the Trail, the slippery Trail and the 4×3 world as `FiniteMDP` objects, evaluate policies exactly with the linear system (3.9) and by summing the series of Proposition 3.5, find optimal policies for a range of discount factors, compute occupancy measures, and check the performance difference lemma of Section 3.9 numerically.

TRY IT INTERACTIVELY

Module 2 of the interactive course (`Apps_claude/RL Course/index.html`) has sliders for p_0, p_1 and γ: drag them and watch the Trail's state values, action values and advantages change, and compare a policy of your own with the optimal one.

3.8 Existence of optimal policies ●●

This section proves the facts about optimality stated in Section 3.7. The proof is short once we know two facts about the optimality equation that Chapter 4 proves: it has a solution, and only one.

Write the right-hand side of the optimality equation as an operator on value functions: for any $V \in \mathbb{R}^{|\mathcal{S}|}$,

$$(\mathcal{T}^* V)(s) = \max_a \Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s') \Big].$$

The optimality equation (3.10) says that V^* is a *fixed point* of $\mathcal{T}^*$, a function with $\mathcal{T}^* V = V$. Theorem 4.3 shows that $\mathcal{T}^*$ is a γ-contraction in the maximum norm, and Banach's fixed-point theorem (Theorem 4.4) then shows that $\mathcal{T}^*$ has exactly one fixed point. Neither result uses anything from this section, so there is no circularity.

Theorem 3.7 Optimal policies exist and can be deterministic and stationary

Let $(\mathcal{S}, \mathcal{A}, P, r, \gamma, d_0)$ be a finite MDP with $\gamma \in [0,1)$ and bounded rewards, and let $\bar{V}$ be the unique solution of $V = \mathcal{T}^* V$. Then:

(a) $V^* = \bar{V}$; that is, the optimal value function is the unique solution of the Bellman optimality equation (3.10).

(b) Every deterministic stationary policy π that is greedy with respect to $\bar{V}$ is optimal: $V^{\pi}(s) \geq V^{\pi'}(s)$ for every state s and every policy π', including history-dependent and randomized policies.

(c) A stationary policy π is optimal if and only if $\mathcal{T}^* V^{\pi} = V^{\pi}$.

Proof. *Step 1: a greedy policy attains $\bar{V}$.* Let π be deterministic, stationary and greedy with respect to $\bar{V}$. In every state,

$$r(s,\pi(s)) + \gamma \sum_{s'} P(s' \mid s,\pi(s))\, \bar{V}(s') = \max_a \Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, \bar{V}(s') \Big] = \bar{V}(s),$$

so $\bar{V}$ solves the Bellman expectation equation of π. By Proposition 3.5 that equation has only one solution, V^{π}; hence $V^{\pi} = \bar{V}$.

Step 2: no policy does better than $\bar{V}$. Let π' be any policy, possibly history-dependent and randomized, and start it in a fixed state $s_0 = s$. At time t, given the history h_t, it draws a_t from some distribution $\pi'_t(\cdot \mid h_t)$. By the Markov property in Definition 3.1,

$$\mathbb{E}\big[r_t + \gamma \bar{V}(s_{t+1}) \mid h_t \big] = \sum_a \pi'_t(a \mid h_t) \Big[r(s_t,a) + \gamma \sum_{s'} P(s' \mid s_t,a)\, \bar{V}(s') \Big] \leq \bar{V}(s_t),$$

because an average of the brackets is at most their maximum, which is $(\mathcal{T}^* \bar{V})(s_t) = \bar{V}(s_t)$. Taking expectations and multiplying by γ^t gives $\mathbb{E}[\gamma^t r_t] \leq \mathbb{E}[\gamma^t \bar{V}(s_t)] - \mathbb{E}[\gamma^{t+1} \bar{V}(s_{t+1})]$. Summing over $t = 0, \dots, n-1$, the right-hand side telescopes:

$$\mathbb{E}\Big[\sum_{t=0}^{n-1} \gamma^t r_t \Big] \leq \bar{V}(s) - \gamma^n\, \mathbb{E}\big[\bar{V}(s_n) \big].$$

As $n \to \infty$, the left-hand side tends to $V^{\pi'}(s)$ (the rewards are bounded, so the tail of the sum is at most $\gamma^n r_{\max}/(1-\gamma)$) and the last term tends to 0 ($\bar{V}$ is bounded). Hence $V^{\pi'}(s) \le \bar{V}(s)$.

Step 3: conclusion. By Step 2, $V^*(s) = \sup_{\pi'} V^{\pi'}(s) \le \bar{V}(s)$; by Step 1, $\bar{V}(s) = V^{\pi}(s) \le V^*(s)$. So $V^* = \bar{V} = V^{\pi}$, which proves (a) and (b).

Step 4: part (c). If π is optimal, then $V^{\pi} = V^* = \bar{V}$ and $\mathcal{T}^* V^{\pi} = \mathcal{T}^* \bar{V} = \bar{V} = V^{\pi}$. Conversely, if $\mathcal{T}^* V^{\pi} = V^{\pi}$, then V^{π} solves the optimality equation, so $V^{\pi} = \bar{V} = V^*$ by uniqueness and (a). ■

Why history and randomization do not help. Step 2 contains the whole reason. Whatever the history, a randomized choice of action yields an average of the one-step values $r(s_t, a) + \gamma \sum_{s'} P(s' \mid s_t, a)\bar{V}(s')$, and an average is never larger than the largest term. The Markov property guarantees that the single number $\bar{V}(s_{t+1})$ summarizes everything that can happen after s_{t+1}, so remembering the past cannot reveal opportunities that $\bar{V}$ misses. Both ingredients matter: the state must be Markov and fully observed, and there must be a single objective.

When randomization does help. Each of the following breaks one of these ingredients.

- *Constraints.* Suppose the agent must maximize reward while keeping an expected cost below a budget. In a one-state problem where action A earns reward 1 at cost 1 and action B earns 0 at cost 0, with a budget of half the maximal discounted cost, the best stationary policy plays A with probability $\frac{1}{2}$; neither deterministic stationary policy is both feasible and optimal. Optimal policies of constrained MDPs may need to randomize (Altman 1999); see Section 3.10.3 and Chapter 23.
- *Opponents.* In matching pennies or rock–paper–scissors, every deterministic strategy can be exploited by an opponent who knows it, and the equilibrium strategies randomize (Chapter 21).
- *Partial observability.* When the agent sees only an observation rather than the state, a randomized policy that depends only on the current observation can do strictly better than every deterministic one that does (Chapter 19).

Two further limits of the theorem are worth noting. In a *finite-horizon* problem the optimal policy usually depends on the time remaining, so it is deterministic and Markov but not stationary (Section 3.10). And with $\gamma = 1$, Step 2 needs $\mathbb{E}[\bar{V}(s_n)] \to 0$, which holds when every policy reaches a terminal state, and the existence and uniqueness of $\bar{V}$ need separate arguments; this is the theory of *stochastic shortest-path problems* (Bertsekas and Tsitsiklis 1996, ch. 2; Bertsekas 2012).

3.9 Occupancy measures and the performance difference lemma ●●

Value functions look forward: from a given state, how much reward lies ahead? *Occupancy measures* look at the same process from the other side: over the whole run, how much time does a policy spend in each state? The two views meet in the *performance difference lemma*, a

formula for how much better one policy is than another. It is the starting point of the trust-region methods of Chapter 12 and of the theory of policy optimization in Chapter 14, and it reappears in offline reinforcement learning and imitation learning (Chapter 17, Chapter 18).

Definition 3.8 Discounted occupancy measure

Let π be a stationary policy and $s_0 \sim d_0$. The *discounted state occupancy* (or discounted state-visitation distribution) of π is

$$d^\pi(s) = (1-\gamma)\sum_{t=0}^{\infty} \gamma^t \Pr_\pi(s_t = s),$$

and its *state–action occupancy* is $d^\pi(s,a) = d^\pi(s)\,\pi(a \mid s) = (1-\gamma)\sum_t \gamma^t \Pr_\pi(s_t = s, a_t = a)$. When the initial distribution matters we write $d^\pi_{d_0}$, or d^π_s for a start in the fixed state s.

Proposition 3.9 Properties of occupancy measures

For every stationary policy π:

(a) d^π is a probability distribution over $\mathcal{S}$.

(b) In matrix form, $(d^\pi)^\top = (1-\gamma)\, d_0^\top (I - \gamma P^\pi)^{-1}$; equivalently, d^π is the unique solution of the *flow equation*

$$d^\pi(s') = (1-\gamma)\, d_0(s') + \gamma \sum_s d^\pi(s)\, P^\pi(s,s') \quad \text{for every } s'. \tag{3.12}$$

(c) The objective $J(\pi) = \sum_s d_0(s) V^\pi(s)$ is an average of rewards under the occupancy measure:

$$J(\pi) = \frac{1}{1-\gamma}\sum_{s,a} d^\pi(s,a)\, r(s,a) = \frac{1}{1-\gamma}\, \mathbb{E}_{s\sim d^\pi,\, a\sim\pi(\cdot|s)}\big[r(s,a)\big]. \tag{3.13}$$

Proof. For (a), the entries are nonnegative and $\sum_s d^\pi(s) = (1-\gamma)\sum_t \gamma^t = 1$. For (b), the distribution of s_t is the row vector $d_0^\top (P^\pi)^t$, so $(d^\pi)^\top = (1-\gamma)\, d_0^\top \sum_t \gamma^t (P^\pi)^t = (1-\gamma)\, d_0^\top (I - \gamma P^\pi)^{-1}$ by Proposition 3.5. Multiplying on the right by $I - \gamma P^\pi$ gives the flow equation, whose solution is unique because the matrix is invertible. For (c), $J(\pi) = d_0^\top V^\pi = d_0^\top (I-\gamma P^\pi)^{-1} r^\pi = \frac{1}{1-\gamma}(d^\pi)^\top r^\pi$, and $r^\pi(s) = \sum_a \pi(a \mid s) r(s,a)$. ∎

Two interpretations help. First, $d^\pi(s)/(1-\gamma) = \sum_t \gamma^t \Pr_\pi(s_t = s)$ is the *expected discounted number of visits* to s. Second, d^π is the distribution of the state s_T at a random time T, drawn independently of the process with $\Pr(T = t) = (1-\gamma)\gamma^t$: it is where the process is when a coin with probability $1-\gamma$ of "stop" first comes up "stop". This gives a simple way to sample from d^π, which policy-gradient methods need (Chapter 11).

On the Trail with the uniform policy and $\gamma = 0.9$, the hiker is in Start at time 0, in Middle at time 1 with probability $\frac{1}{2}$, and in Goal at time 2 with probability $\frac{1}{4}$. So

$$d^\pi(\text{Start}) = 0.1, \qquad d^\pi(\text{Middle}) = 0.1{\times}0.9{\times}\tfrac{1}{2} = 0.045, \qquad d^\pi(\text{Goal}) = 0.1{\times}0.9^2{\times}\tfrac{1}{4} \approx 0.020,$$

(0.02025 exactly), and the remaining 0.83475 is on End, the probability of having already finished. Since rewards in End are zero, it does not contribute to (3.13): $J(\pi) = 10 \times (0.1 \times 0.5 + 0.045 \times 1 + 0.02025 \times 4) = 10 \times 0.176 = 1.76$. For the optimal policy, $d^{\pi^*} = (0.1, 0.09, 0.081, 0.729)$.

PITFALL

The discounted occupancy d^π is not the stationary distribution of the Markov chain P^π. It depends on the initial distribution and on γ, and in episodic tasks it weights early states more than late ones. Averaging over all states visited along sampled trajectories, without the weights γ^t, gives a different distribution; Section 11.9 explains what policy-gradient implementations that do this actually optimize.

Lemma 3.10 Performance difference lemma

For any two stationary policies π and π' and any initial distribution d_0,

$$J(\pi') - J(\pi) = \frac{1}{1-\gamma}\, \mathbb{E}_{s\sim d^{\pi'},\, a\sim\pi'(\cdot|s)}\big[A^\pi(s,a)\big],$$

where $d^{\pi'}$ is the discounted state occupancy of π' started from d_0 and A^π is the advantage function of π. In particular, for every state s, $V^{\pi'}(s) - V^\pi(s) = \frac{1}{1-\gamma}\mathbb{E}_{\tilde{s}\sim d^{\pi'}_s,\, a\sim\pi'(\cdot|\tilde{s})}[A^\pi(\tilde{s},a)]$.

Proof. Generate $s_0, a_0, r_0, s_1, \ldots$ by following π' from $s_0 \sim d_0$. Then $J(\pi') = \mathbb{E}_{\pi'}[\sum_t \gamma^t r_t]$ and $J(\pi) = \mathbb{E}[V^\pi(s_0)]$. For every n, the sum

$$\sum_{t=0}^{n-1} \gamma^t \big[r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)\big] = \sum_{t=0}^{n-1} \gamma^t r_t + \gamma^n V^\pi(s_n) - V^\pi(s_0)$$

telescopes. Take expectations under π' and let $n \to \infty$; because rewards and V^π are bounded, the right-hand side tends to $J(\pi') - J(\pi)$. On the left, condition on (s_t, a_t): by the Markov property and (3.6),

$$\mathbb{E}_{\pi'}\big[r_t + \gamma V^\pi(s_{t+1}) \mid s_t, a_t\big] = r(s_t,a_t) + \gamma \sum_{s'} P(s' \mid s_t, a_t)\, V^\pi(s') = Q^\pi(s_t,a_t),$$

so the t-th term has expectation $\gamma^t\, \mathbb{E}_{\pi'}[A^\pi(s_t,a_t)]$. Therefore

$$J(\pi') - J(\pi) = \sum_{t=0}^{\infty} \gamma^t \sum_s \Pr_{\pi'}(s_t = s) \sum_a \pi'(a \mid s)\, A^\pi(s,a) = \frac{1}{1-\gamma} \sum_s d^{\pi'}(s) \sum_a \pi'(a \mid s)\, A^\pi(s,a).$$

The exchange of the sums over t and s is allowed because $|A^\pi| \le 2r_{\max}/(1-\gamma)$ and $\sum_t \gamma^t$ converges. The per-state version is the case $d_0 = \delta_s$. ■

Read the lemma from right to left: to find out how much better π' is than π, run π', and at every state it visits, score its action by how much better that action is than π's average there, *according to π's values*. The states come from the new policy, the advantages from the old one. This asymmetry is the source of much of the difficulty of policy optimization: we can usually

estimate the advantages of the policy we have been running, but we have no samples from the occupancy of the policy we are thinking of switching to.

Example 3.5 The performance difference lemma on the Trail

Let π be the uniform policy and π' the optimal one (go, go) on the Trail with $\gamma = 0.9$. Directly, $J(\pi') - J(\pi) = 3.24 - 1.76 = 1.48$. For the lemma we need the advantages of π for the actions of π': $A^\pi(\text{Start}, \text{go}) = 2.52 - 1.76 = 0.76$, $A^\pi(\text{Middle}, \text{go}) = 3.6 - 2.8 = 0.8$, and $A^\pi = 0$ in Goal (one action) and in End. With $d^{\pi'} = (0.1, 0.09, 0.081, 0.729)$,

$$\frac{1}{1-\gamma}\sum_s d^{\pi'}(s) A^\pi(s, \pi'(s)) = 10 \times (0.1 \times 0.76 + 0.09 \times 0.8) = 10 \times 0.148 = 1.48.$$

If we used the old policy's occupancy $d^\pi = (0.1, 0.045, \ldots)$ instead, we would get $10 \times (0.076 + 0.045 \times 0.8) = 1.12$, which underestimates the improvement: π' reaches Middle twice as often as π does. This second quantity is the improvement predicted by the *surrogate objective* of trust-region methods ((12.1)); it agrees with the true difference to first order when π' is close to π.

♦

An immediate consequence: if $\sum_a \pi'(a \mid s) A^\pi(s, a) \geq 0$ in every state, every term in the lemma is nonnegative, so $V^{\pi'} \geq V^\pi$ for every starting distribution. A policy that never does worse than π's average according to π's own advantages is at least as good as π. This is the *policy improvement theorem*, which Chapter 4 proves directly (Theorem 4.6) and turns into an algorithm. The lemma in this form is due to Kakade and Langford (2002); Agarwal et al. (2022) give a modern treatment.

3.10 Other objectives ●●

The discounted infinite-horizon return is the main objective of this book, but it is not the only sensible one. This section describes three alternatives: a fixed number of decisions, the long-run average reward, and a reformulation of the whole problem as a linear program.

3.10.1 Finite-horizon problems

Some tasks have a fixed number H of decisions: a game of a set number of rounds, a trading day, a language model that writes at most H tokens. The return is $\sum_{t=0}^{H-1} \gamma^t r_t$, often with $\gamma = 1$. The value of a state now depends on how many decisions remain, so values and optimal policies carry a time index: $V_t^*(s)$ is the best expected return from state s at time t. They satisfy a time-indexed optimality equation that can be solved *backward*, starting from the end,

$$V_H^*(s) = 0, \qquad V_t^*(s) = \max_a \Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s, a)\, V_{t+1}^*(s') \Big], \quad t = H-1, \ldots, 0, \tag{3.14}$$

and an optimal policy chooses at time t an action that attains the maximum. This procedure, *backward induction*, is the simplest dynamic-programming algorithm, and Chapter 4 returns to it.

The optimal policy of a finite-horizon problem generally depends on time. On the Trail with $\gamma = 0.9$, suppose only H decisions are allowed. With one decision left, exiting is best in both Start and Middle ($V^*_{H-1} = 1$ and 2), because going on would end the episode with nothing. With two decisions left, $V^*_{H-2}(\text{Start}) = \max\{1, 0.9 \times 2\} = 1.8$ and $V^*_{H-2}(\text{Middle}) = \max\{2, 0.9 \times 4\} = 3.6$: going on is best in both. With three left, $V^*_{H-3}(\text{Start}) = \max\{1, 0.9{\times}3.6\} = 3.24$. The optimal rule, "go on if at least two decisions remain, otherwise exit", is not stationary. Adding the number of remaining decisions to the state makes it stationary again, at the price of a larger state space.

The same observation explains a classic bug. An episodic task that is cut off by a time limit is a finite-horizon problem; if the time remaining is not part of the state, the problem is no longer Markov, and learning algorithms must treat the cut-off differently from a genuine termination (Section 6.8). Conversely, for large H and $\gamma < 1$ the horizon barely matters: Chapter 4 shows that V^*_0 differs from the infinite-horizon V^* by at most $\gamma^H r_{\max}/(1-\gamma)$.

3.10.2 The average-reward criterion ●●●

In some continuing tasks no discounting is natural: a router or a factory is judged by its long-run throughput, and a reward a thousand steps from now matters as much as one now. The *average-reward* or *gain* criterion measures a stationary policy by

$$g^\pi(s) = \lim_{n\to\infty} \frac{1}{n}\, \mathbb{E}_\pi\Big[\sum_{t=0}^{n-1} r_t \mid s_0 = s\Big].$$

For finite MDPs this limit exists for every stationary policy. If the Markov chain P^π is *unichain*, meaning it has a single recurrent class (plus possibly transient states), then the gain does not depend on the start: $g^\pi = \sum_s \mu^\pi(s)\, r^\pi(s)$, where μ^π is the stationary distribution of P^π. The gain says nothing about what happens early on; that information is in the *bias* $h^\pi(s)$, the total amount by which rewards from s exceed the average (formally a Cesàro limit of $\mathbb{E}_\pi[\sum_{t<n}(r_t - g^\pi) \mid s_0 = s]$). Gain and bias satisfy the *differential* Bellman equation

$$g^\pi + h^\pi(s) = \sum_a \pi(a \mid s)\Big[r(s,a) + \sum_{s'} P(s' \mid s,a)\, h^\pi(s')\Big] \quad \text{for every } s, \tag{3.15}$$

which determines h^π up to an additive constant; one fixes it, for example, by requiring $\sum_s \mu^\pi(s) h^\pi(s) = 0$. When every deterministic stationary policy is unichain (and more generally when the MDP is *weakly communicating*), the optimality equation $g^* + h^*(s) = \max_a[r(s,a) + \sum_{s'} P(s' \mid s,a) h^*(s')]$ has a solution, and a policy that attains the maximum in every state has the largest gain (Puterman 1994, ch. 8–9). Without such assumptions the optimal gain can depend on the starting state, and the theory becomes more delicate.

The discounted and average-reward criteria are closely related. For a stationary unichain policy, $V^\pi_\gamma(s) = g^\pi/(1-\gamma) + h^\pi(s) + o(1)$ as $\gamma \to 1$, with h^π normalized as above (Puterman 1994, sec. 8.2): the discounted value is dominated by the gain, and the bias is the first correction. Blackwell (1962) showed that some deterministic stationary policy is optimal for *all* discount factors in an interval $(\gamma_0, 1)$; such *Blackwell-optimal* policies also maximize the gain.

A two-state example makes this concrete. In state 0 the action stay earns 1 and switch moves to state 1 with reward 0; in state 1, stay earns 2 and switch moves back to state 0 with reward 0.

The policy "switch in 0, stay in 1" ends up in state 1 forever: it is unichain with gain 2, and its bias is $h(1) = 0$ and $h(0) = -2$, the reward lost by spending one step in state 0. "Stay in 0, switch in 1" has gain 1; "switch in both" alternates and has gain 0; "stay in both" is not unichain, and its gain is 1 or 2 depending on where it starts. The gain-optimal policy is therefore "switch in 0, stay in 1". Under discounting, switching in state 0 is optimal exactly when $2\gamma/(1-\gamma) > 1/(1-\gamma)$, that is when $\gamma > \frac{1}{2}$; so the Blackwell-optimal policy is optimal for all $\gamma \in (\frac{1}{2}, 1)$. And indeed $V_\gamma(0) = 2\gamma/(1-\gamma) = 2/(1-\gamma) - 2 = g/(1-\gamma) + h(0)$.

3.10.3 The linear-programming view ●●●

The optimality equation is nonlinear because of the maximum, but the whole problem can be written as a *linear program* (LP), an observation due to Manne (1960) and d'Epenoux (1963). Fix positive weights $w(s) > 0$ that sum to 1, for example $w = d_0$ when every state can be a starting state. The *primal LP* looks for the smallest value function that satisfies all the one-step inequalities:

$$\min_{V \in \mathbb{R}^{|\mathcal{S}|}} \sum_s w(s)\, V(s) \quad \text{subject to} \quad V(s) \geq r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s') \quad \text{for all } s, a. \tag{3.16}$$

Every feasible V satisfies $V \geq \mathcal{T}^* V$, and Chapter 4 shows that this forces $V \geq V^*$; since V^* itself is feasible, it is the unique solution. The *dual LP* has one variable $d(s,a) \geq 0$ for each state–action pair:

$$\begin{aligned} &\max_{d \geq 0} \ \frac{1}{1-\gamma} \sum_{s,a} d(s,a)\, r(s,a) \\ &\text{subject to} \quad \sum_a d(s',a) = (1-\gamma)\, w(s') + \gamma \sum_{s,a} P(s' \mid s,a)\, d(s,a) \quad \text{for all } s'. \end{aligned} \tag{3.17}$$

The constraints are exactly the flow equation (3.12) with $d_0 = w$, and the objective is (3.13). So every occupancy measure of a stationary policy is feasible. Conversely, every feasible d is the occupancy measure of the stationary policy $\pi_d(a \mid s) = d(s,a) / \sum_{a'} d(s,a')$ (any distribution where the denominator is zero): writing $d(s,a) = \mu(s)\, \pi_d(a \mid s)$ with $\mu(s) = \sum_a d(s,a)$ turns the constraint into the flow equation of π_d for μ, whose solution is unique by Proposition 3.9; so $\mu = d^{\pi_d}$, and d is the state–action occupancy of π_d. The dual LP is therefore the problem "choose the best achievable occupancy measure", and the set of achievable occupancy measures is a polytope. By LP duality both programs have the same optimal value, $\sum_s w(s) V^*(s)$; the vertices of the polytope correspond to deterministic policies, and Chapter 4 shows how complementary slackness reads off an optimal policy from an optimal d.

This view matters beyond computation, because a linear objective over occupancy measures is easy to extend. Adding a constraint $\frac{1}{1-\gamma} \sum_{s,a} d(s,a)\, c(s,a) \leq C$ on an expected discounted cost gives the LP of a *constrained MDP*, whose solutions may randomize, as the example in Section 3.8 showed (Altman 1999); Chapter 23 uses Lagrangian methods built on this formulation. In offline reinforcement learning, ratios of occupancy measures $d^\pi(s,a)/d^{\mathcal{D}}(s,a)$ between a target policy and the data are estimated through this duality (Chapter 17); and imitation learning can be phrased as matching the occupancy measure of an expert (Chapter 18).

3.11 What MDPs leave out

The MDP is a deliberately simple model, and it is worth being clear about what it does not capture.

Hidden state. The model assumes the agent sees a Markov state. A robot with noisy sensors, a doctor who cannot observe a disease directly, or a card player who cannot see the other hands sees only observations, and the best action depends on what can be inferred from the history of observations. These are *partially observable* MDPs (Chapter 19).

Other agents. The transition probabilities are fixed. When other learning agents share the environment, its behavior changes as they learn, and from each agent's point of view the world is no longer a fixed MDP. Games and multi-agent learning are the subject of Chapter 21.

The reward. The model takes the reward as given. In real applications someone must design it, and a reward that is slightly wrong can produce behavior that is very wrong (Chapter 23); sometimes it is easier to learn the reward from demonstrations or preferences (Chapter 18). Constraints and risk, which do not fit a single expected sum, need extensions such as the constrained MDPs of Section 3.10.3.

Size. Nothing in the definition limits the number of states, but the methods of this chapter and the next store one number per state. Problems with large or continuous state spaces need function approximation (Part III).

Choosing a model. When you formulate a real problem as an MDP, four choices matter most. The *state* should contain whatever affects the future (stock on hand *and* orders in transit; position *and* velocity), and as little else as possible, since every irrelevant variable multiplies the number of states. The *actions* should be at a time scale where decisions matter: a robot that chooses a new torque every millisecond faces a much longer credit-assignment problem than one that chooses among short movements (Chapter 20). The *reward* should measure the outcome you care about rather than your guess about how to achieve it. And the *discount factor* should reflect how far ahead the agent needs to look: in an episodic task with a natural end, $\gamma = 1$ or close to it; in a continuing task, a γ whose effective horizon $1/(1-\gamma)$ covers the delays between actions and their consequences. Values such as $\gamma = 0.99$, common in deep reinforcement learning, correspond to a horizon of about 100 steps.

HISTORICAL NOTE

Andrey Markov introduced the chains that bear his name in 1906. Sequential decision problems in this form were studied by Lloyd Shapley in his 1953 paper on stochastic games, whose one-player case is an MDP (Shapley 1953), and by Richard Bellman, who introduced the term "Markovian decision process" and the principle of optimality in the 1950s (Bellman 1957a, 1957b). Ronald Howard's book on dynamic programming and Markov processes made the model widely known (Howard 1960), and the linear-programming formulation followed in the early 1960s (Manne 1960).

3.12 Summary

- A *Markov decision process* consists of states, actions, transition probabilities $P(s' \mid s, a)$, expected rewards $r(s, a)$, a discount factor γ and an initial distribution. Its defining assumption, the *Markov property*, is that the current state and action determine the distribution of what happens next; it is a property of the state we choose to describe the problem.
- Episodic tasks fit the same model by treating terminal states as absorbing states with zero reward. The *slippery Trail*, in which each go fails with probability 0.2, is our running example of a problem with loops.
- A *policy* maps states to (probabilities of) actions. The *return* $G_t = \sum_k \gamma^k r_{t+k}$ satisfies $G_t = r_t + \gamma G_{t+1}$; the discount factor sets an effective horizon of $1/(1-\gamma)$ and is part of the problem.
- The value functions V^π, Q^π and A^π are expected returns. They satisfy the *Bellman expectation equations*, a linear system with the unique solution $V^\pi = (I - \gamma P^\pi)^{-1} r^\pi$. On the Trail with the uniform policy, $V^\pi(\text{Middle}) = 2.8$ and $V^\pi(\text{Start}) = 1.76$; on the slippery Trail, 2.681 and 1.610.
- The optimal values satisfy the *Bellman optimality equations*, with a maximum in place of the policy's average. Every finite discounted MDP has an optimal policy that is deterministic and stationary: any policy that is greedy with respect to V^* (●● proof in Section 3.8).
- The Trail's optimal policy goes all the way when $\gamma > 0.5$ and exits at once when $\gamma < 0.5$; on the slippery Trail the threshold moves to 5/9.
- (●●) The *discounted occupancy measure* d^π says where a policy spends its time, and $J(\pi) = \frac{1}{1-\gamma}\mathbb{E}_{d^\pi}[r]$. The *performance difference lemma* writes $J(\pi') - J(\pi)$ as the average advantage of π's values under π''s occupancy.
- (●●●) Finite-horizon problems have time-dependent optimal policies computed by backward induction; the average-reward criterion is the limit of discounting as $\gamma \to 1$; and the whole problem can be written as a pair of linear programs, the dual one over occupancy measures.

The next chapter turns these equations into algorithms: dynamic programming computes V^π, V^* and optimal policies when the model is known, and its theory explains why the learning methods of the rest of the book work.

3.13 Further Reading

Puterman (1994) is the standard reference on MDPs: finite-horizon problems (ch. 4), discounted problems (ch. 6, including the linear program in sec. 6.9) and average reward (ch. 8–9), all with complete proofs. Bertsekas (2012) covers the same ground from the viewpoint of control and adds a thorough treatment of stochastic shortest-path problems and approximate dynamic programming. Chapter 3 of Sutton and Barto (2018) introduces MDPs gently with many examples, using the R_{t+1} reward convention. Szepesvári (2010) gives a concise and rigorous summary, and the first chapter of Agarwal et al. (2022) states the facts used in modern theory, including occupancy measures and the performance difference lemma, with clean proofs.

The original sources are readable. Bellman (1957b) introduced the principle of optimality and

the functional equations; Howard (1960) introduced policy iteration (the subject of Chapter 4). The performance difference lemma in its modern form is from Kakade and Langford (2002), whose conservative policy iteration started the line of work that led to TRPO and PPO (Chapter 12). The 4×3 world comes from Russell and Norvig (2021), whose chapter on sequential decisions is a good second view of this material.

For the alternatives to discounting, Blackwell (1962) is the classic paper on discount factors close to one, and Mahadevan (1996) surveys average-reward reinforcement learning. For constrained MDPs and the role of the linear program, see Altman (1999). The debate about whether a scalar reward is enough is in Silver et al. (2021) and the works cited in Chapter 1.

(●●●) Average-reward methods are attracting renewed interest because discounting with γ close to 1 makes learning slow and unstable; Wan et al. (2021) give learning and planning algorithms for the average-reward setting with convergence guarantees. Occupancy-measure formulations underlie much recent theory, from the analysis of policy gradients (Chapter 14) to offline reinforcement learning (Chapter 17).

3.14 Exercises

Exercise 3.1 ● Trail values for several policies

For each of the following policies and discount factors, compute $V^\pi(\text{Middle})$, $V^\pi(\text{Start})$, $Q^\pi(\text{Start}, \text{go})$ and $A^\pi(\text{Start}, \text{go})$ with the Bellman equation: (a) $p_0 = 0.8$, $p_1 = 0.5$, $\gamma = 0.9$; (b) $p_0 = 1$, $p_1 = 0.5$, $\gamma = 0.8$; (c) $p_0 = 0.5$, $p_1 = 1$, $\gamma = 0.6$. In case (b), why is $A^\pi(\text{Start}, \text{go}) = 0$?

Exercise 3.2 ● The slippery Trail with any slip probability

On a slippery Trail where each go fails with probability q, consider the policy that always goes. (a) Write its Bellman equations and solve them for $V^\pi(\text{Middle})$ and $V^\pi(\text{Start})$ as functions of q and γ. (b) Evaluate them for $\gamma = 0.9$ and $q = 0.2$ and $q = 0.5$. (c) For $\gamma = 0.9$, up to which slip probability is going on still optimal? (d) What is the expected number of steps of an episode under this policy when $q = 0.2$?

Exercise 3.3 ● Is it Markov?

Say whether each description is a Markov state for the stated task. If it is not, propose a small addition that makes it Markov. (a) The position of a car on a road, for a cruise controller. (b) The cards in a blackjack player's hand and the dealer's visible card, when cards are dealt from a fresh shuffled deck after every hand, and when they are dealt from the same deck without reshuffling. (c) The stock level in the inventory problem of Example 3.1 when deliveries take two days. (d) The hiker's station on the slippery Trail when every episode is stopped after three decisions. Why is the station enough on the plain Trail with the same time limit?

Exercise 3.4 ● The return recursion and a bound on values

Let rewards be bounded by $r_{\max}$. (a) Prove $G_t = r_t + \gamma G_{t+1}$ from the definition (3.3). (b) Show that $|V^\pi(s)| \le r_{\max}/(1-\gamma)$ for every policy and state, and give an MDP in which the bound holds with equality. (c) On the slippery Trail with $\gamma = 0.9$, compare the bound with the actual largest value.

Exercise 3.5 • From Q to V and back

On the slippery Trail with the uniform policy and $\gamma = 0.9$, the action values are $Q^\pi(\text{Start}, \cdot) = (1, 2.2204)$ and $Q^\pi(\text{Middle}, \cdot) = (2, 3.3626)$ for (exit, go). (a) Compute $V^\pi(\text{Start})$ and $V^\pi(\text{Middle})$ from these. (b) Using only V^π and the model, recompute $Q^\pi(\text{Middle}, \text{go})$. (c) Check that the advantages in each state average to zero under π. (d) Which of the two directions, $Q \to V$ or $V \to Q$, needs the model? Why does this matter for learning without a model?

Exercise 3.6 • Modeling an inventory problem

Complete the model of Example 3.1: list $P(\cdot \mid s, a)$ and $r(s, a)$ for all six state–action pairs. Then solve it with `rlcourse` for $\gamma = 0.9$ and describe the optimal ordering policy in words. How would you change the state if deliveries took two days?

Exercise 3.7 • Counting policies

Counting shows why brute force fails. (a) How many deterministic stationary policies does the 4×3 world have (count only the nine cells where a decision is made)? (b) How many does the inventory problem of Example 3.1 have? (c) Explain why enumerating all deterministic policies and evaluating each is a correct but impractical way to solve an MDP, and estimate how long it would take for the 4×3 world if each evaluation took a millisecond.

Exercise 3.8 •• Shifting all rewards

This exercise explains the pitfall in Section 3.4. (a) In a continuing MDP, add a constant c to every reward. Show that every value $V^\pi(s)$ increases by $c/(1-\gamma)$ and that the optimal policies do not change. (b) On the Trail with $\gamma = 0.9$, add c to the reward of every action taken in Start, Middle and Goal (but not in End). Find the values of c for which exiting at once is optimal. (c) Explain the difference between (a) and (b) in terms of the absorbing terminal state.

Exercise 3.9 •• Invertibility of $I - \gamma P^\pi$

This exercise gives a second proof of Proposition 3.5 and looks at $\gamma = 1$. (a) Show that every eigenvalue λ of a stochastic matrix satisfies $|\lambda| \le 1$, and deduce again that $I - \gamma P^\pi$ is invertible for $\gamma < 1$. (b) Show that the row sums of $(I - \gamma P^\pi)^{-1}$ equal $1/(1-\gamma)$ by applying it to the vector of ones. (c) For the Trail with $\gamma = 1$, show that $I - P^\pi$ is singular for every policy. (d) Show that the 3×3 system for the non-terminal states alone is invertible when $\gamma = 1$, and use it to compute the uniform policy's undiscounted values.

Exercise 3.10 ●● The performance difference lemma at work

On the Trail with $\gamma = 0.9$, let π be the uniform policy and π' the policy that goes on in Start and exits in Middle. (a) Compute $J(\pi') - J(\pi)$ directly and with the lemma. (b) Compute the same difference with d^{π} in place of $d^{\pi'}$ and compare. (c) Verify the per-state version of the lemma for a start in Middle. (d) Using the lemma, prove: if $\sum_a \pi'(a \mid s) A^{\pi}(s,a) \geq 0$ in every state, then $V^{\pi'}(s) \geq V^{\pi}(s)$ in every state.

Exercise 3.11 ●● Occupancy measures of the Trail

Occupancy measures can be computed by hand on small problems. (a) Find d^{π} on the Trail for a general policy (p_0, p_1) and discount γ, and verify (3.13) against the formula for $J(p_0, p_1)$. (b) Solve the flow equation (3.12) by hand for the uniform policy on the slippery Trail with $\gamma = 0.9$, and recompute $J(\pi)$ from it. (c) Describe a simulation that estimates d^{π} without knowing the model, and say how many episodes you would need to estimate $d^{\pi}(\text{Goal})$ to within ± 0.005.

Exercise 3.12 ●● A history-dependent policy

On the slippery Trail with $\gamma = 0.9$, consider the policy "in Start, go on unless you have already slipped there in this episode, in which case exit; in Middle, always go". (a) Explain why this policy is not Markov in the original states. (b) Compute its value from Start by adding a flag "has slipped" to the state. (c) Compare with $V^*(\text{Start})$ and relate the result to Theorem 3.7.

Exercise 3.13 ●● The slippery Trail with a deadline

On the slippery Trail with $\gamma = 0.9$, suppose the hiker may make at most H decisions. (a) Use backward induction (3.14) to compute the optimal value of Start for $H = 1, 2, 3$, and the optimal first action. (b) Show that $V_0^*(\text{Start})$ increases with H and converges to $V^*(\text{Start}) \approx 3.084$. (c) In what sense is the optimal policy for a fixed H non-stationary?

Exercise 3.14 ●●● Gain and bias of a two-state MDP

Use the two-state MDP of Section 3.10.2. (a) For the randomized policy that stays with probability $\frac{1}{2}$ in both states, find the stationary distribution, the gain and the bias (normalized so that its stationary average is zero), and check the differential Bellman equation. (b) Compute its discounted values for $\gamma = 0.9$, 0.99 and 0.999 and compare them with $g/(1-\gamma) + h$. (c) The policy "stay in both states" is not unichain. What is its gain from each state, and why does the differential equation with a single gain have no solution for it? (d) Read the discussion of average-reward learning in Wan et al. (2021) and explain one difficulty that does not arise with discounting.

Exercise 3.15 ●●● The two linear programs of a small MDP

For the two-state MDP of Section 3.10.2 with $\gamma = 0.9$ and weights $w = (\frac{1}{2}, \frac{1}{2})$: (a) write the primal LP (3.16) with its four constraints and the dual LP (3.17) with its four variables; (b) find the optimal dual solution by computing the occupancy measure of the optimal policy, and check

that both programs have the same optimal value; (c) verify complementary slackness: the dual variables that are positive belong to constraints that hold with equality. Which constraints are slack, and what does that say about the corresponding actions?

Exercise 3.16 ●●● When the best policy must randomize

A one-state continuing MDP has two actions: A earns reward 1 and incurs cost 1; B earns 0 and costs 0. With $\gamma = 0.9$, the agent must keep its expected discounted cost at most 5. (a) Write this constrained problem as the dual LP over occupancy measures with one extra constraint, and solve it. (b) Show that no deterministic stationary policy is optimal. (c) Form the Lagrangian with multiplier $\lambda \geq 0$ for the cost constraint. Show that for the optimal multiplier every stationary policy maximizes the Lagrangian, so that the Lagrangian alone does not pick out the optimal policy; what does this imply for Lagrangian methods in practice (Chapter 23)? (d) Is there a deterministic but non-stationary policy that is optimal? Discuss, and look up what Altman (1999) says about the number of states in which an optimal stationary policy must randomize.

CHAPTER 4

Dynamic Programming

WHAT YOU WILL LEARN

- Explain what dynamic programming computes, what it assumes, and why the learning methods of later chapters are approximations of it.
- Evaluate a policy by iterative sweeps, synchronously or in place, and choose a sound stopping rule.
- Prove that the Bellman operators are monotone γ-contractions and derive the error bounds that follow from Banach's fixed-point theorem.
- Prove the policy improvement theorem and use it to show that policy iteration ends with an optimal policy.
- Run value iteration with a stopping rule that guarantees a near-optimal policy, and explain generalized policy iteration.
- Bound the loss of a greedy policy and of approximate value and policy iteration (●●).
- Analyze the cost of dynamic programming, and relate it to linear programming and to the linear-quadratic regulator (●● and ●●●).

Chapter 3 reduced the evaluation of a policy and the search for an optimal one to equations: the Bellman expectation equation for V^π and the Bellman optimality equation for V^*. This chapter is about solving them. When the model of the environment, the transition probabilities P and the expected rewards r, is known, the equations can be solved by computation alone, without any interaction. The family of algorithms that does this by repeatedly applying the Bellman equations as update rules is called *dynamic programming* (DP), a name Richard Bellman gave it in the 1950s (Bellman 1957b).

Dynamic programming matters even though most of this book is about problems in which the model is *not* known. Almost every method in later chapters is an approximation of a DP algorithm. Monte Carlo and temporal-difference methods replace the expectations over P by averages over sampled experience; deep reinforcement learning replaces the table of values by a neural network; actor-critic methods and PPO alternate between evaluating and improving a policy, exactly as policy iteration does. The theory of this chapter, above all the contraction property of the Bellman operators and the policy improvement theorem, is what explains why those methods work and how they can fail.

We start with policy evaluation by repeated sweeps (Section 4.2) and the theory behind it: Bellman operators, contraction and Banach's fixed-point theorem (Section 4.3). Then come the two classic control algorithms, policy iteration (Section 4.4) and value iteration (Section 4.5), and the view that unites them and all of reinforcement learning, generalized policy iteration (Sec-

tion 4.6). Section 4.10 covers finite horizons. These sections form the Foundations track. The ●● sections prove the stopping rule of value iteration and Banach's theorem, bound the loss of acting greedily on an imperfect value function (Section 4.7), analyze the cost of DP (Section 4.8) and solve the linear-quadratic regulator (Section 4.10.1); the ●●● sections discuss strongly polynomial bounds and the linear-programming method (Section 4.9).

4.1 Planning with a known model

Throughout this chapter the MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma, d_0)$ of Definition 3.1 is finite and *known*: we can look up $P(s' \mid s, a)$ and $r(s, a)$ for every state, action and next state. Computing a good policy from such a model, without interacting with the environment, is called *planning*. There are two planning problems.

- Prediction (policy evaluation): given a policy π, compute its value function V^π.
- Control: find an optimal policy π^* and the optimal value function V^*.

Chapter 3 already solved the prediction problem exactly by solving a linear system, at a cost that grows like $|\mathcal{S}|^3$. Dynamic programming takes a different route. It starts from a guess and improves it by *sweeps*: in one sweep, every state's value is replaced by the right-hand side of a Bellman equation evaluated at the current guess. A sweep costs at most $|\mathcal{S}|^2|\mathcal{A}|$ operations, far fewer when each state has only a few possible successors, and the guesses converge to the solution geometrically fast. This makes DP practical for problems with millions of states, as long as the states can be enumerated and the model is known.

The main assumption, a known model, is strong. A chess program does know the rules, and an inventory manager may have a good model of demand; a robot learning to walk usually has no such model, and a language model's "environment" is a person. Parts II to VII of the book are about what to do then. But the algorithms of this chapter remain the blueprint, and they are used directly whenever a model is available or has been learned (Chapter 8, Chapter 15).

4.2 Policy evaluation

The Bellman expectation equation (3.7) says that V^π is consistent with itself: the value of each state equals its expected reward plus the discounted expected value of the next state. *Iterative policy evaluation* turns this consistency condition into an update rule. Start with any guess V_0, for example $V_0 = 0$, and repeatedly replace every value by the right-hand side of the equation:

$$V_{k+1}(s) = \sum_a \pi(a \mid s)\Big[r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V_k(s')\Big] \quad \text{for every } s. \tag{4.1}$$

Each such update is called a *backup*, because it carries value back from successor states to their predecessors, and updating every state once is a *sweep*. Section 4.3 proves that $V_k \to V^\pi$ from every starting point.

Algorithm 4.1 Iterative policy evaluation

Input: a finite MDP (P, r, γ), a policy π, a tolerance $\theta > 0$, an initial guess V (for example $V = 0$; terminal states must start at 0).

1. Repeat:
   1. $\Delta \leftarrow 0$.
   2. For each state s:
      1. $v \leftarrow \sum_a \pi(a \mid s)\big[r(s,a) + \gamma \sum_{s'} P(s' \mid s, a)\, V(s')\big]$.
      2. $\Delta \leftarrow \max\{\Delta, |v - V(s)|\}$.
      3. Store v as the new value of s: in the *synchronous* version in a separate array V_{new}, which replaces V at the end of the sweep; in the *in-place* version directly in $V(s)$.
2. until $\Delta < \theta$.

Output: $V \approx V^\pi$. For the synchronous version, $\|V - V^\pi\|_\infty < \gamma\theta/(1-\gamma)$.

Here $\|V\|_\infty = \max_s |V(s)|$ is the *maximum norm*, the largest absolute entry of a vector, so $\|V - V^\pi\|_\infty$ is the worst error over all states. The error guarantee in the output line follows from the contraction property (Corollary 4.5).

Example 4.1 Evaluating the uniform policy on the Trail

Take the uniform policy on the Trail with $\gamma = 0.9$ and start from $V_0 = 0$. In each sweep, $V_{k+1}(\text{Goal}) = 4$, $V_{k+1}(\text{Middle}) = \frac{1}{2} \times 2 + \frac{1}{2} \times 0.9\, V_k(\text{Goal})$ and $V_{k+1}(\text{Start}) = \frac{1}{2} \times 1 + \frac{1}{2} \times 0.9\, V_k(\text{Middle})$. Table 4.1 shows the synchronous sweeps.

Table 4.1. Synchronous iterative policy evaluation of the uniform policy on the Trail ($\gamma = 0.9$, $V_0 = 0$).

sweep k	V_k(Start)	V_k(Middle)	V_k(Goal)	largest change Δ
0	0	0	0	
1	0.5	1	4	4
2	0.95	2.8	4	1.8
3	1.76	2.8	4	0.81
4	1.76	2.8	4	0

Value flows backward one station per sweep: after the first sweep only Goal knows about its +4; after the second, Middle has learned of it; after the third, Start. In general, starting from $V_0 = 0$, $V_k(s)$ is the expected discounted reward of the first k steps from s. Every episode of the Trail ends within three steps, so $V_3 = V^\pi$ exactly and the fourth sweep changes nothing.

The in-place version can be faster, and the order of the sweep matters. Sweeping in the order Goal, Middle, Start, each state uses values that were updated earlier in the same sweep:

$V(\text{Goal}) = 4$, then $V(\text{Middle}) = 1 + 0.45 \times 4 = 2.8$, then $V(\text{Start}) = 0.5 + 0.45 \times 2.8 = 1.76$. The exact answer arrives after a single sweep. In the forward order, in place behaves exactly like the synchronous version here.

♦

On the slippery Trail no finite number of sweeps gives the exact answer, because value circulates around the loops. Table 4.2 shows what happens instead: the error shrinks geometrically. After the first few sweeps it falls by roughly a factor of 10 per sweep, and eight sweeps bring it below 10^{-5}.

Table 4.2. Synchronous evaluation of the uniform policy on the slippery Trail ($\gamma = 0.9$, $V_0 = 0$; $V_k(\text{Goal}) = 4$ for $k \geq 1$).

sweep k	$V_k(\text{Start})$	$V_k(\text{Middle})$	error $\lVert V_k - V^\pi \rVert_\infty$
1	0.5	1	1.68
2	0.905	2.53	0.705
3	1.49225	2.66770	0.118
4	1.59467	2.68009	0.0155
5	1.60835	2.68121	0.00184
8	1.61019	2.68132	0.0000023
exact	1.61019	2.68132	0

The factor of 10 is no accident. Once Goal is right, the error in Middle is multiplied at each sweep by $\gamma \times 0.1 = 0.09$, the discounted probability of staying in Middle under the uniform policy, and the error in Start follows the same pattern. The general guarantee of the next section is weaker, a factor of $\gamma = 0.9$ per sweep; actual convergence is often much faster than the guarantee.

When to stop. In practice one stops when the largest change Δ in a sweep falls below a tolerance θ. A small change does not by itself mean a small error, though. The bound in Algorithm 4.1 is $\gamma\theta/(1-\gamma)$: with $\gamma = 0.99$ and $\theta = 0.001$, a sweep that changes no value by more than 0.001 is compatible with an error of up to 0.099. Choose θ with the effective horizon $1/(1-\gamma)$ in mind.

PITFALL

Terminal states must have value 0 and must stay at 0. If the initial guess gives the terminal state a nonzero value, a synchronous sweep will propagate it, because the terminal state's own backup ($r = 0$, then stay) multiplies it by γ each sweep; it decays only geometrically, slowing convergence and contaminating every value computed meanwhile. Code that stores terminal states as absorbing (as `rlcourse` does) should initialize them to zero, or skip them in the sweep.

4.3 Bellman operators and contraction

Why does iterative evaluation converge, from any starting point, and how fast? The answer is a single geometric fact about the Bellman equations, and the same fact explains value iteration,

Q-learning (Chapter 6), and much more. To state it we treat a whole sweep as one mathematical object.

A sweep takes a value function $V \in \mathbb{R}^{|\mathcal{S}|}$ and returns a new one. For a stationary policy π, the *Bellman expectation operator* $\mathcal{T}^\pi$ and the *Bellman optimality operator* $\mathcal{T}^*$ are the maps

$$(\mathcal{T}^\pi V)(s) = \sum_a \pi(a \mid s)\Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s')\Big],$$
$$(\mathcal{T}^* V)(s) = \max_a \Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s')\Big].$$

In the matrix notation of Section 3.6, $\mathcal{T}^\pi V = r^\pi + \gamma P^\pi V$. A synchronous sweep of policy evaluation is $V_{k+1} = \mathcal{T}^\pi V_k$; the Bellman equations say that V^π is a *fixed point* of $\mathcal{T}^\pi$ ($\mathcal{T}^\pi V^\pi = V^\pi$) and V^* is a fixed point of $\mathcal{T}^*$. The same definitions apply to action-value functions, with $(\mathcal{T}^\pi Q)(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \sum_{a'} \pi(a' \mid s') Q(s',a')$ and $(\mathcal{T}^* Q)(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a')$.

Two simple properties of these operators carry most of the theory. The first is monotonicity: better inputs give better outputs. The second is how they treat constants.

Proposition 4.1 Monotonicity and constant shifts

Let $U, W \in \mathbb{R}^{|\mathcal{S}|}$ with $U \le W$ in every state, let c be a number and $\mathbf{1}$ the vector of ones. Then $\mathcal{T}^\pi U \le \mathcal{T}^\pi W$ and $\mathcal{T}^* U \le \mathcal{T}^* W$; and $\mathcal{T}^\pi(U + c\mathbf{1}) = \mathcal{T}^\pi U + \gamma c \mathbf{1}$ and $\mathcal{T}^*(U + c\mathbf{1}) = \mathcal{T}^* U + \gamma c \mathbf{1}$.

Proof. $\mathcal{T}^\pi W - \mathcal{T}^\pi U = \gamma P^\pi (W - U)$, and P^π has nonnegative entries, so a nonnegative vector stays nonnegative. For $\mathcal{T}^*$, each bracket $r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s')$ is larger for W than for U, so the maximum over a is too. Adding c to every value adds γc to every bracket, because the probabilities $P(s' \mid s,a)$ sum to one; averages and maxima pass the constant through. ∎

For $\mathcal{T}^*$ we also need an elementary inequality about maxima.

Lemma 4.2 Maxima are non-expansive

For any two functions f and g on a finite set $\mathcal{A}$, $\left|\max_a f(a) - \max_a g(a)\right| \le \max_a |f(a) - g(a)|$.

Proof. Let a^* maximize f. Then $\max_a f(a) - \max_a g(a) \le f(a^*) - g(a^*) \le \max_a |f(a) - g(a)|$. Exchanging the roles of f and g bounds the difference from the other side. ∎

Theorem 4.3 Bellman operators are γ-contractions

For every stationary policy π and all $U, W \in \mathbb{R}^{|\mathcal{S}|}$,

$$\|\mathcal{T}^\pi U - \mathcal{T}^\pi W\|_\infty \le \gamma \, \|U - W\|_\infty \qquad \text{and} \qquad \|\mathcal{T}^* U - \mathcal{T}^* W\|_\infty \le \gamma \, \|U - W\|_\infty .$$

The same holds for the action-value operators, with the maximum norm taken over all state–action pairs.

Proof. For $\mathcal{T}^\pi$ and any state s,

$$\left|(\mathcal{T}^\pi U)(s) - (\mathcal{T}^\pi W)(s)\right| = \gamma \left|\sum_{s'} P^\pi(s,s')\,\big(U(s') - W(s')\big)\right| \le \gamma \sum_{s'} P^\pi(s,s')\,\|U - W\|_\infty = \gamma\,\|U - W\|_\infty,$$

because the row $P^\pi(s,\cdot)$ is a probability distribution. For $\mathcal{T}^*$, apply Lemma 4.2 with $f(a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)U(s')$ and $g(a)$ the same expression with W:

$$\left|(\mathcal{T}^* U)(s) - (\mathcal{T}^* W)(s)\right| \le \max_a \gamma \left|\sum_{s'} P(s' \mid s,a)\big(U(s') - W(s')\big)\right| \le \gamma\,\|U - W\|_\infty.$$

Taking the maximum over s proves both inequalities. The action-value versions are proved the same way, with $\sum_{a'} \pi(a' \mid s')Q(s',a')$ or $\max_{a'} Q(s',a')$ in place of $V(s')$. ■

In words: one sweep brings any two value functions at least γ times closer together, measured in the worst state. The reason is that every backup averages the next-state values, which cannot increase their spread, and then multiplies the result by γ; the maximum over actions does not spoil this. Figure 4.1 shows the effect on the slippery Trail: two value functions that start far apart are driven together, and both are driven toward V^*, which, being a fixed point, does not move.

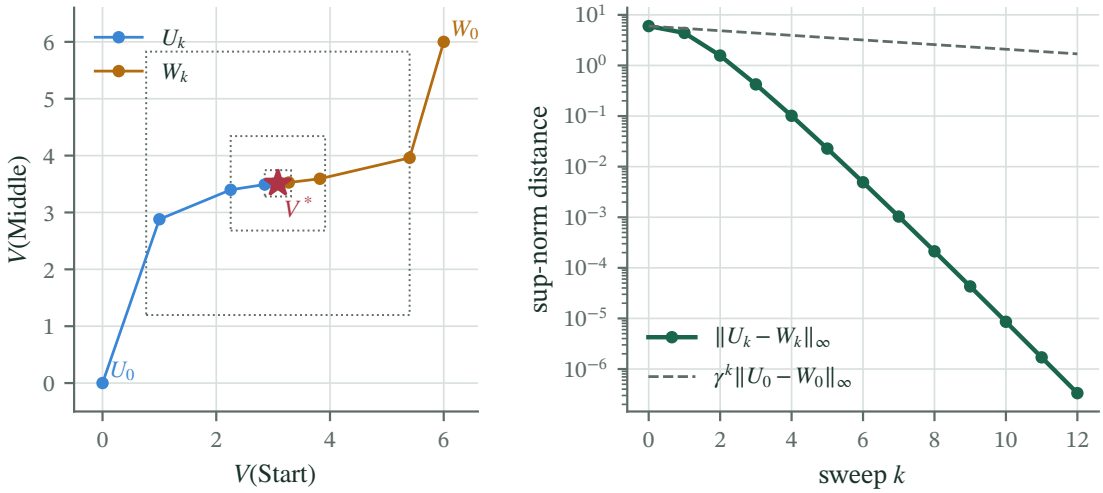


Figure 4.1. Value iteration on the slippery Trail ($\gamma = 0.9$) from two starting value functions U_0 and W_0; since $V(\text{Goal}) = 4$ is fixed, each value function is a point in the plane. Left: both sequences approach V^* (star); the dotted squares are the smallest maximum-norm balls around V^* that contain U_k and W_k for $k = 1, 2, 3$. Right: the distance $\|U_k - W_k\|_\infty$ shrinks at least as fast as the guarantee $\gamma^k\|U_0 - W_0\|_\infty$ (dashed), here much faster.

A map that brings points closer by a fixed factor has exactly one fixed point, and iterating it from anywhere finds that point. This is Banach's fixed-point theorem, one of the most useful results in analysis. We state it for vectors with any norm, which is all we need; Section 4.3.1 proves it, and Theorem A.2 in Appendix A states it for general complete metric spaces.

Theorem 4.4 Banach's fixed-point theorem

Let $\|\cdot\|$ be a norm on $\mathbb{R}^n$ and let $F : \mathbb{R}^n \to \mathbb{R}^n$ satisfy $\|F(U) - F(W)\| \le \gamma \|U - W\|$ for all U, W, with a constant $\gamma \in [0, 1)$. Then:

(a) F has exactly one fixed point $\bar{V}$.

(b) For every starting point V_0, the sequence $V_{k+1} = F(V_k)$ converges to $\bar{V}$, and $\|V_k - \bar{V}\| \le \gamma^k \|V_0 - \bar{V}\|$.

(c) For every $k \ge 1$, $\|V_k - \bar{V}\| \le \frac{\gamma}{1-\gamma} \|V_k - V_{k-1}\|$ and $\|V_k - \bar{V}\| \le \frac{\gamma^k}{1-\gamma} \|V_1 - V_0\|$.

Applied to the Bellman operators, the two theorems give everything we need about the convergence of DP.

Corollary 4.5 Convergence of policy evaluation and value iteration

In a finite MDP with $\gamma < 1$:

(a) V^π is the unique fixed point of $\mathcal{T}^\pi$, and V^* is the unique fixed point of $\mathcal{T}^*$.

(b) From any V_0, iterative policy evaluation ($V_{k+1} = \mathcal{T}^\pi V_k$) and value iteration ($V_{k+1} = \mathcal{T}^* V_k$) converge, with $\|V_k - V^\pi\|_\infty \le \gamma^k \|V_0 - V^\pi\|_\infty$ and $\|V_k - V^*\|_\infty \le \gamma^k \|V_0 - V^*\|_\infty$.

(c) If the last sweep changed no value by more than θ, that is $\|V_k - V_{k-1}\|_\infty \le \theta$, then the current values are within $\gamma\theta/(1-\gamma)$ of the fixed point.

Part (a) for $\mathcal{T}^*$ is the fact that Theorem 3.7 relied on: the optimality equation has exactly one solution. Part (b) is an *a priori* bound: it tells us in advance how many sweeps suffice. Part (c) is an *a posteriori* bound: it certifies the accuracy of what we have computed, using only the size of the last change. The rate γ is a worst case; Table 4.2 showed a much faster rate, and Section 4.8 discusses when the worst case is attained.

> **PITFALL**
>
> The contraction holds in the *maximum* norm. In the ordinary Euclidean norm, $\mathcal{T}^\pi$ need not be a contraction at all (Exercise 4.9 gives a two-state example). This seemingly technical point has large consequences: when values are approximated by a function fitted in a least-squares sense, the fitting step is a non-expansion in a (weighted) Euclidean norm while the Bellman operator is a contraction only in the maximum norm, and their combination can diverge. Chapter 9 studies this "deadly triad".

4.3.1 Proof of Banach's fixed-point theorem ●●

Proof of Banach's fixed-point theorem. *Successive steps shrink.* By the contraction property, $\|V_{k+1} - V_k\| = \|F(V_k) - F(V_{k-1})\| \le \gamma \|V_k - V_{k-1}\|$, so by induction $\|V_{k+1} - V_k\| \le \gamma^k \|V_1 - V_0\|$.

The sequence is Cauchy. For $m > k$, the triangle inequality and the geometric series give

$$\|V_m - V_k\| \le \sum_{j=k}^{m-1} \|V_{j+1} - V_j\| \le \sum_{j=k}^{\infty} \gamma^j \|V_1 - V_0\| = \frac{\gamma^k}{1-\gamma} \|V_1 - V_0\|,$$

which tends to 0 as $k \to \infty$. In $\mathbb{R}^n$ every Cauchy sequence converges (all norms on $\mathbb{R}^n$ are equivalent, and $\mathbb{R}^n$ is complete), so $V_k \to \bar{V}$ for some $\bar{V}$.

The limit is a fixed point. F is continuous, since $\|F(U) - F(W)\| \le \gamma\|U - W\|$. Hence $F(\bar{V}) = \lim_k F(V_k) = \lim_k V_{k+1} = \bar{V}$.

It is the only one. If $F(\bar{W}) = \bar{W}$ as well, then $\|\bar{V} - \bar{W}\| = \|F(\bar{V}) - F(\bar{W})\| \le \gamma\|\bar{V} - \bar{W}\|$, which for $\gamma < 1$ forces $\bar{V} = \bar{W}$. This proves (a).

Rates. $\|V_k - \bar{V}\| = \|F(V_{k-1}) - F(\bar{V})\| \le \gamma\|V_{k-1} - \bar{V}\|$, and induction gives (b). For (c), $\|V_k - \bar{V}\| \le \gamma\|V_{k-1} - \bar{V}\| \le \gamma\big(\|V_{k-1} - V_k\| + \|V_k - \bar{V}\|\big)$; solving for $\|V_k - \bar{V}\|$ gives the first bound, and combining it with $\|V_k - V_{k-1}\| \le \gamma^{k-1}\|V_1 - V_0\|$ gives the second. ∎

The proof never used the structure of value functions, only the contraction inequality. This is why the same theorem appears again and again in this book: for the action-value operators, for the soft Bellman operator of maximum-entropy reinforcement learning (Chapter 13), for the distributional Bellman operator (Chapter 10), and, in a stochastic version, for Q-learning (Chapter 6).

4.4 Policy improvement

Suppose we have evaluated a policy π and know V^π, hence also $Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s')$. How can we find a better policy? A natural idea: in every state, look one step ahead with the values we have and choose the action that looks best, $\pi'(s) \in \arg\max_a Q^\pi(s,a)$. This new policy is *greedy* with respect to V^π. But is it better? Its first action looks good *assuming* that π is followed afterward, whereas π' will in fact use its own greedy choices from then on. The policy improvement theorem says that this can only help.

Theorem 4.6 Policy improvement

Let π and π' be stationary policies such that, in every state s,

$$\sum_a \pi'(a \mid s)\, Q^\pi(s,a) \ge V^\pi(s).$$

Then $V^{\pi'}(s) \ge V^\pi(s)$ in every state s. If the inequality in the assumption is strict in some state s, then $V^{\pi'}(s) > V^\pi(s)$ in that state. In particular, every policy that is greedy with respect to V^π is at least as good as π.

Proof. The assumption says $\mathcal{T}^{\pi'} V^\pi \ge V^\pi$. Apply $\mathcal{T}^{\pi'}$ to both sides and use monotonicity (Proposition 4.1): $(\mathcal{T}^{\pi'})^2 V^\pi \ge \mathcal{T}^{\pi'} V^\pi \ge V^\pi$, and by induction $(\mathcal{T}^{\pi'})^k V^\pi \ge V^\pi$ for every k. By Corollary 4.5, $(\mathcal{T}^{\pi'})^k V^\pi \to V^{\pi'}$ as $k \to \infty$, so $V^{\pi'} \ge V^\pi$.

For the strict part, subtract V^π from both sides of $V^{\pi'} = r^{\pi'} + \gamma P^{\pi'} V^{\pi'}$ and rearrange:

$$(I - \gamma P^{\pi'})(V^{\pi'} - V^\pi) = r^{\pi'} + \gamma P^{\pi'} V^\pi - V^\pi = \mathcal{T}^{\pi'} V^\pi - V^\pi.$$

By the invertibility result of Section 3.6, $(I - \gamma P^{\pi'})^{-1} = \sum_k \gamma^k (P^{\pi'})^k$ has nonnegative entries, and its diagonal entries are at least 1 (the $k = 0$ term is I). So $V^{\pi'}(s) - V^\pi(s) \ge (\mathcal{T}^{\pi'} V^\pi - V^\pi)(s)$, which

is positive if the assumption is strict in s. Finally, a greedy π' satisfies $\sum_a \pi'(a \mid s) Q^\pi(s,a) = \max_a Q^\pi(s,a) \geq V^\pi(s)$, because $V^\pi(s)$ is an average of the $Q^\pi(s,a)$. ■

The idea of the proof in words: taking the greedy action once and then following π is at least as good as following π from the start. Replacing the next step of π by a greedy step is, by the same argument, at least as good again, and so on; in the limit we follow π' forever. (The performance difference lemma, Lemma 3.10, gives another proof: every advantage $A^\pi(s, \pi'(s))$ is nonnegative.)

On the Trail with $\gamma = 0.9$ and the uniform policy, $Q^\pi(\text{Start}, \cdot) = (1, 2.52)$ and $Q^\pi(\text{Middle}, \cdot) = (2, 3.6)$ for (exit, go). The greedy policy goes on in both states, and its values $(3.24, 3.6)$ are higher than the uniform policy's $(1.76, 2.8)$ in every state, as the theorem promises.

When does improvement stop? Suppose the greedy policy π' is no better than π anywhere. Then the strict part of the theorem shows that $\max_a Q^\pi(s,a) = V^\pi(s)$ in every state, that is, $\mathcal{T}^* V^\pi = V^\pi$: the value function of π satisfies the optimality equation, and by Theorem 3.7 π is optimal. So greedy improvement either strictly improves a policy or proves that it is already optimal.

4.4.1 Policy iteration

Alternating evaluation and greedy improvement gives *policy iteration*, introduced by Howard (1960).

Algorithm 4.2 Policy iteration

Input: a finite MDP (P, r, γ) and an initial deterministic policy π_0.

1. $k \leftarrow 0$.
2. Repeat:
   1. *Evaluation:* compute $V_k = V^{\pi_k}$, by solving $(I - \gamma P^{\pi_k}) V = r^{\pi_k}$ or by Algorithm 4.1.
   2. *Improvement:* for each state s, let $Q_k(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_k(s')$ and set $\pi_{k+1}(s) \leftarrow \pi_k(s)$ if $\pi_k(s)$ maximizes $Q_k(s, \cdot)$, and otherwise $\pi_{k+1}(s) \leftarrow$ any maximizer of $Q_k(s, \cdot)$.
   3. If $\pi_{k+1} = \pi_k$: stop. Otherwise $k \leftarrow k+1$.

Output: an optimal policy π_k and its value function $V_k = V^*$.

The tie rule in step 2.2, keep the current action if it is still among the best, looks like a detail, but without it the algorithm can switch forever between equally good actions and never stop (Exercise 4.11). With it, termination is guaranteed.

Corollary 4.7 Policy iteration terminates with an optimal policy

In a finite MDP with $\gamma < 1$, policy iteration stops after at most $|\mathcal{A}|^{|\mathcal{S}|}$ iterations, and the policy it returns is optimal.

Proof. If $\pi_{k+1} \neq \pi_k$, then by the tie rule some state s has $Q_k(s, \pi_{k+1}(s)) > Q_k(s, \pi_k(s)) = V^{\pi_k}(s)$. By Theorem 4.6, $V^{\pi_{k+1}} \geq V^{\pi_k}$ in every state, with strict inequality in s. The value functions therefore increase strictly along the run, so no policy can occur twice. Since there are only $|\mathcal{A}|^{|\mathcal{S}|}$ deterministic stationary policies, the algorithm must stop. When it stops, π_k is greedy with respect to its own values, so $\mathcal{T}^* V^{\pi_k} = \mathcal{T}^{\pi_k} V^{\pi_k} = V^{\pi_k}$, and π_k is optimal by Theorem 3.7. ■

Example 4.2 Policy iteration on the Trail and on the 4×3 world

On the Trail with $\gamma = 0.9$, start from the policy that exits everywhere. *Evaluation:* $V^{\pi_0} = (1, 2, 4)$ for Start, Middle, Goal. *Improvement:* $Q_0(\text{Start}, \text{go}) = 0.9 \times 2 = 1.8 > 1$ and $Q_0(\text{Middle}, \text{go}) = 0.9 \times 4 = 3.6 > 2$, so π_1 goes on in both states. *Evaluation:* $V^{\pi_1} = (3.24, 3.6, 4)$. *Improvement:* exiting now gives $1 < 3.24$ and $2 < 3.6$, so $\pi_2 = \pi_1$ and the algorithm stops with the optimal policy after two evaluations.

Figure 4.2 shows a larger run, on the 4×3 world of Section 3.7 with $\gamma = 0.9$, starting from the poor policy "always move right", which walks into the −1 exit from the middle row. One improvement step already fixes the top and middle rows. Two more steps settle the bottom row, and the fourth policy is greedy for its own values: it is optimal. Four evaluations out of $4^9 = 262{,}144$ possible policies.

♦

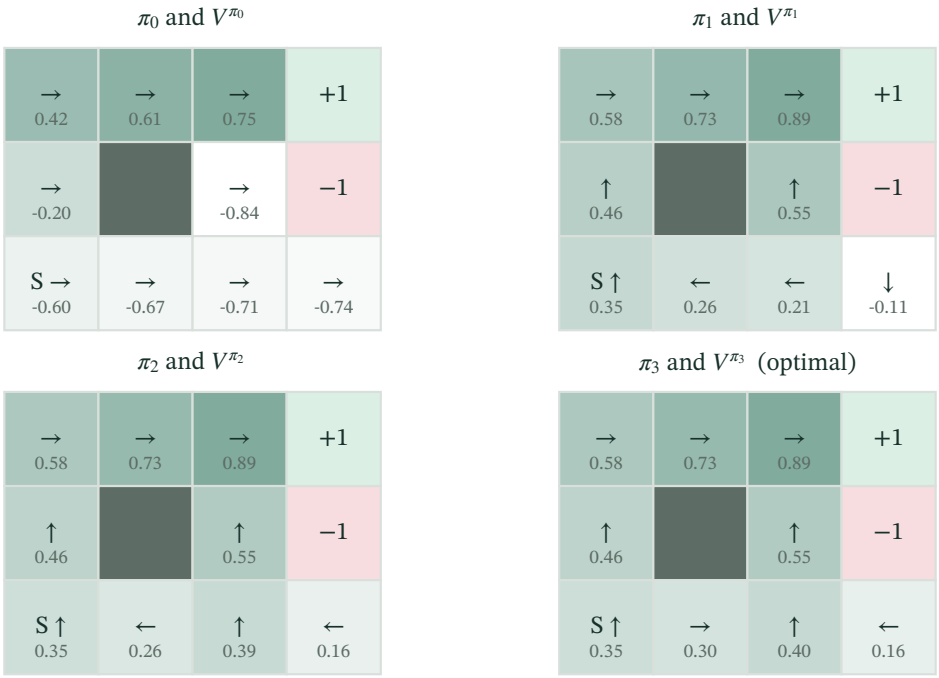


Figure 4.2. Policy iteration on the 4×3 world ($\gamma = 0.9$), starting from "always move right". Each panel shows a policy (arrows) and its values (numbers and shading); the next policy is greedy with respect to these values. The values never decrease from one panel to the next, as the policy improvement theorem guarantees, and the fourth policy is optimal.

In practice policy iteration needs remarkably few iterations, typically a handful even for large problems, far below the bound $|\mathcal{A}|^{|\mathcal{S}|}$ (Section 4.8). Each iteration, however, includes a full policy evaluation, which is expensive for large state spaces. That cost motivates value iteration.

4.5 Value iteration

Policy iteration evaluates each policy exactly before improving it. Is that necessary? On the Trail, after a single sweep of evaluation from $V = 0$ the greedy policy is already optimal; the remaining sweeps only polish values that no longer change the decision. *Value iteration* takes this to the extreme: it performs one sweep of the optimality operator,

$$V_{k+1}(s) = \max_a \Big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V_k(s') \Big] \quad \text{for every } s, \tag{4.2}$$

that is, $V_{k+1} = \mathcal{T}^* V_k$. Two views of this update are useful. It is fixed-point iteration for $\mathcal{T}^*$, so by Corollary 4.5 it converges to V^* from any start, with error at most $\gamma^k \|V_0 - V^*\|_\infty$. And it is *truncated policy iteration*: each sweep improves the policy greedily (the maximum) and then evaluates it by a single backup instead of solving for its value.

Algorithm 4.3 Value iteration

Input: a finite MDP (P, r, γ) with $0 < \gamma < 1$, an accuracy $\varepsilon > 0$, an initial V_0 (for example $V_0 = 0$).

1. $k \leftarrow 0$.
2. Repeat:
   1. For each state s: $V_{k+1}(s) \leftarrow \max_a \big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V_k(s') \big]$.
   2. $k \leftarrow k + 1$.
3. until $\|V_k - V_{k-1}\|_\infty < \varepsilon(1-\gamma)/(2\gamma)$.
4. For each state s: $\pi(s) \leftarrow$ any maximizer of $r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V_k(s')$.

Output: a policy π with $\|V^* - V^\pi\|_\infty < \varepsilon$, and values V_k with $\|V_k - V^*\|_\infty < \varepsilon/2$.

The stopping rule in step 3 rests on a guarantee proved in Section 4.5.1: the policy that is greedy with respect to V_k satisfies

$$\|V^* - V^{\pi_k}\|_\infty \le \frac{2\gamma}{1-\gamma} \|V_k - V_{k-1}\|_\infty. \tag{4.3}$$

The computed values need not be accurate for the policy to be good. Note also what the output does *not* promise: the policy may be optimal long before the rule fires, and the rule cannot detect that.

Example 4.3 Value iteration on the Trail and the slippery Trail

Table 4.3 runs value iteration from $V_0 = 0$ with $\gamma = 0.9$. On the Trail, the first sweep gives each state its best immediate reward, $V_1 = (1, 2, 4)$; the second lets Middle see Goal, $V_2(\text{Middle}) = \max\{2, 0.9 \times 4\} = 3.6$; the third lets Start see it too, $V_3(\text{Start}) = \max\{1, 0.9 \times 3.6\} = 3.24$, and $V_3 = V^*$. On the slippery Trail the values approach $V^*(\text{Start}) \approx 3.0839$ and $V^*(\text{Middle}) \approx 3.5122$ geometrically; after the first few sweeps the error falls by a factor of four to five per sweep.

Table 4.3. Value iteration from $V_0 = 0$ with $\gamma = 0.9$; $V_k(\text{Goal}) = 4$ for $k \geq 1$. The error column is $\|V_k - V^*\|_\infty$ for the slippery Trail.

sweep k	Trail Start	Trail Middle	slippery Start	slippery Middle	slippery error
1	1	2	1	2	2.084
2	1.8	3.6	1.62	3.24	1.464
3	3.24	3.6	2.6244	3.4632	0.459
4	3.24	3.6	2.9659	3.5034	0.118
5			3.0563	3.5106	0.028
9			3.0838	3.5122	0.00006

On both versions of the Trail, the greedy policy for V_1 is already optimal: from V_1, going on is worth $0.9 \times 2 = 1.8 > 1$ in Start of the plain Trail, and $0.9\,(0.8 \times 2 + 0.2 \times 1) = 1.62 > 1$ on the slippery one. With $\varepsilon = 0.01$ the stopping threshold is $0.01 \times 0.1/1.8 \approx 0.00056$; the plain Trail stops after sweep 4, whose change is 0, and the slippery Trail after sweep 9, whose largest change is 0.00022.

♦

The 4×3 world shows the same pattern on a larger scale. Figure 4.3 shows the values after 1, 2, 3, 5 and 10 sweeps: value spreads backward from the +1 exit, about one cell per sweep, the way it spread from Goal to Start on the Trail. Figure 4.4 measures the convergence. Its left panel compares the error of value iteration with the guarantee $\gamma^k \|V_0 - V^*\|_\infty$ and with policy iteration; its right panel shows the loss of the greedy policy after each sweep. The greedy policy is optimal from the third sweep on, while the values need about twenty sweeps in all to become accurate to 10^{-6}, and the stopping-rule bound (4.3) is far above the true loss all along. *In-place* value iteration, which uses each new value as soon as it is computed, converges in noticeably fewer sweeps.

Policy iteration or value iteration? Policy iteration needs few iterations, but each one solves a linear system (or runs many evaluation sweeps). Value iteration needs many iterations, about $\log(1/\varepsilon)/(1-\gamma)$ in the worst case, but each costs only one sweep. For small problems the difference hardly matters. For large ones the best choice is usually in between, which leads to the idea of the next section.

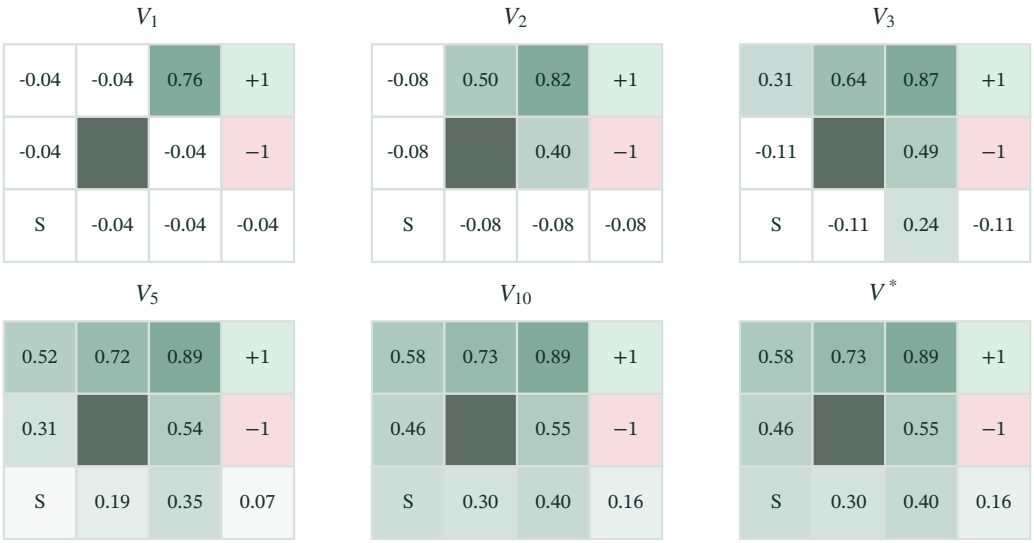


Figure 4.3. Value iteration on the 4×3 world ($\gamma = 0.9$) from $V_0 = 0$: the values after k sweeps, and the optimal values V^*. The +1 exit's value spreads backward through the grid, a cell or so per sweep; after ten sweeps they are within 0.01 of V^*.

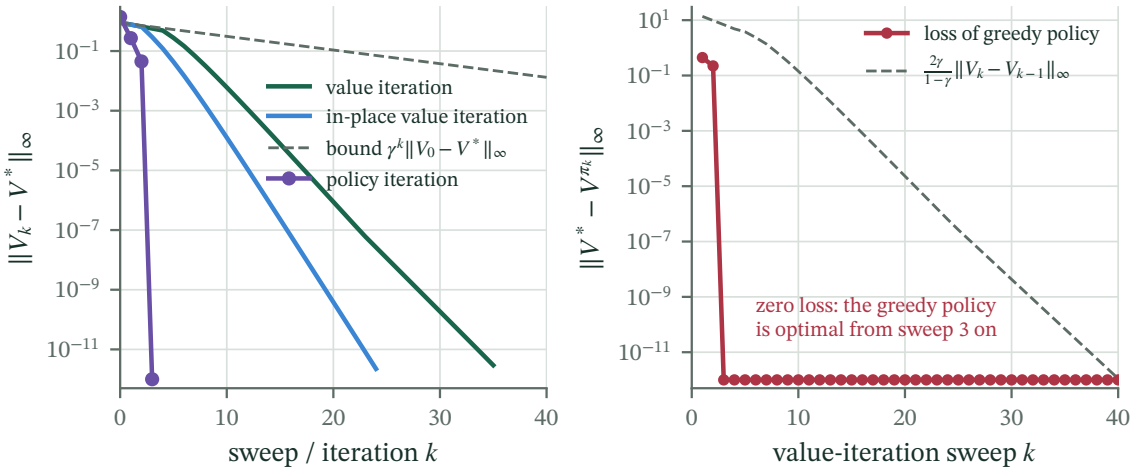


Figure 4.4. Convergence on the 4×3 world ($\gamma = 0.9$). Left: the error $\|V_k - V^*\|_\infty$ of synchronous and in-place value iteration, the guarantee $\gamma^k\|V_0 - V^*\|_\infty$ (dashed), and the values of policy iteration's policies, which reach V^* after three improvements. Right: the loss of the greedy policy after k value-iteration sweeps (zero from the third sweep on, drawn at the bottom of the axis) and the stopping-rule bound $\frac{2\gamma}{1-\gamma}\|V_k - V_{k-1}\|_\infty$ (dashed).

CHECKPOINT

1. In Table 4.3, why does value iteration on the plain Trail reach V^* after exactly three sweeps?
2. Value iteration has just made a sweep in which no value changed by more than 0.001, with $\gamma = 0.99$. What can you say about the greedy policy?
3. Policy iteration's policies improve at every step. Do value iteration's values?

Answers.

1. Starting from $V_0 = 0$, V_k is the best expected reward obtainable in k steps, and every episode of the Trail ends within three steps.
2. By (4.3) its loss is at most $2 \times 0.99/0.01 \times 0.001 = 0.198$ in every state. It may well be optimal, but the rule cannot tell.
3. Not necessarily. From $V_0 = 0$ with nonnegative rewards they increase monotonically (Exercise 4.8), but in general V_k need not be the value of any policy, and it can move up or down.

4.5.1 Why the stopping rule works ●●

Proposition 4.8 Stopping rule for value iteration

Let $V_k = \mathcal{T}^* V_{k-1}$, let π_k be greedy with respect to V_k, and let $\delta = \|V_k - V_{k-1}\|_\infty$. Then

$$\|V_k - V^*\|_\infty \le \frac{\gamma\,\delta}{1-\gamma}, \qquad \|V^{\pi_k} - V_k\|_\infty \le \frac{\gamma\,\delta}{1-\gamma}, \qquad \|V^* - V^{\pi_k}\|_\infty \le \frac{2\gamma\,\delta}{1-\gamma}.$$

In particular, if $\delta < \varepsilon(1-\gamma)/(2\gamma)$, then π_k is ε-optimal and V_k is within $\varepsilon/2$ of V^*.

Proof. The first bound is part (c) of Theorem 4.4 for $F = \mathcal{T}^*$. For the second, greediness means $\mathcal{T}^{\pi_k} V_k = \mathcal{T}^* V_k$, and $V_k = \mathcal{T}^* V_{k-1}$, so

$$\|V^{\pi_k} - V_k\|_\infty \le \|\mathcal{T}^{\pi_k} V^{\pi_k} - \mathcal{T}^{\pi_k} V_k\|_\infty + \|\mathcal{T}^* V_k - \mathcal{T}^* V_{k-1}\|_\infty \le \gamma \|V^{\pi_k} - V_k\|_\infty + \gamma\delta,$$

using $V^{\pi_k} = \mathcal{T}^{\pi_k} V^{\pi_k}$ and the contraction property twice. Solving for $\|V^{\pi_k} - V_k\|_\infty$ gives the second bound, and the triangle inequality $\|V^* - V^{\pi_k}\|_\infty \le \|V^* - V_k\|_\infty + \|V_k - V^{\pi_k}\|_\infty$ gives the third. ■

This stopping rule is standard (Puterman 1994, ch. 6); Williams and Baird (1993) studied bounds of this kind for greedy policies and showed that they are tight in the worst case. In typical problems the bound is conservative, as the right panel of Figure 4.4 shows. The proof uses only the last sweep, so with in-place sweeps one can finish with a single synchronous sweep and apply the proposition to it.

4.6 Generalized policy iteration

Policy iteration and value iteration are two extremes of one idea. Both maintain a policy and a value function and alternate between two processes:

- *evaluation*, which makes the value function consistent with the current policy (moving V toward V^π), and

- *improvement*, which makes the policy greedy with respect to the current value function.

Policy iteration completes each process before starting the other. Value iteration interleaves them as finely as possible: one backup of evaluation per improvement. Any other interleaving works too, as long as both processes keep acting on all states. Sutton and Barto (2018) call this family *generalized policy iteration* (GPI). Figure 4.5 pictures it. The two processes pull in different directions: making the policy greedy usually makes the value function inaccurate for it, and evaluating the policy usually reveals that it is no longer greedy. Yet they have a joint goal, a policy that is greedy with respect to its own value function, and at that point both are satisfied: the value function is V^* and the policy is optimal.

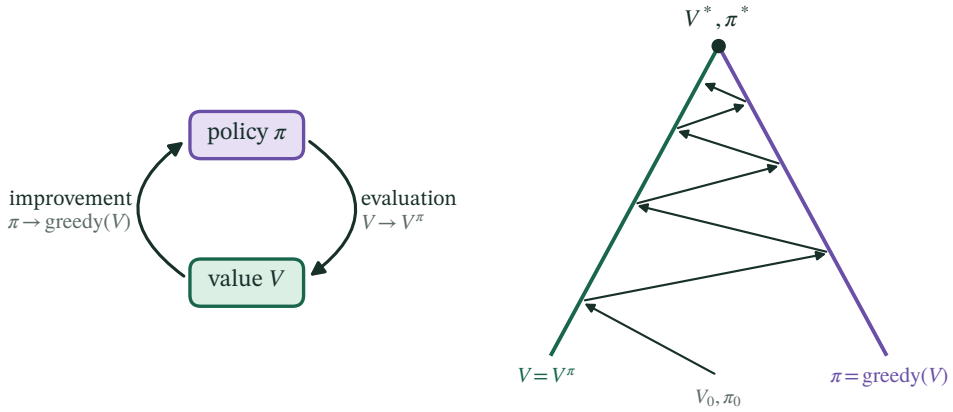


Figure 4.5. Generalized policy iteration. Left: evaluation and improvement act alternately on a policy and a value function. Right: each process moves toward its own goal (the two lines), and alternating between them, with complete or partial steps, ends where both goals hold: the optimal value function and an optimal policy.

Modified policy iteration. The simplest member of the family between the two extremes is *modified policy iteration* (Puterman and Shin 1978): after each greedy improvement, apply $\mathcal{T}^\pi$ a fixed number m of times instead of evaluating π exactly. With $m = 1$ this is value iteration; as $m \to \infty$ it becomes policy iteration. On the 4×3 world with $\gamma = 0.9$ and a target accuracy of 10^{-6}, $m = 1$ needs 24 improvements (and 24 sweeps), $m = 5$ needs 7 improvements and 31 sweeps, and $m = 20$ needs 4 improvements but 61 sweeps. Since an improvement costs $|\mathcal{A}|$ times as much as an evaluation sweep, a moderate m is often the cheapest choice overall (Exercise 4.6).

Asynchronous dynamic programming. Sweeps need not cover all states in a fixed order. *Asynchronous* DP updates states one at a time, in any order, using the latest values of the others. It still converges to V^π or V^* as long as every state is updated infinitely often (Bertsekas and Tsitsiklis 1996). This freedom makes it possible to concentrate computation where it matters: on states the agent actually visits, or on states whose values have just changed. Real-time dynamic programming and prioritized sweeping (Section 8.4.2 and Chapter 8) are built on it.

The bridge to learning. Almost every method in the rest of this book is a form of GPI. Monte Carlo control (Chapter 5) evaluates a policy by averaging sampled returns and improves it greedily. SARSA and Q-learning (Chapter 6) perform a single sampled backup of evaluation between implicit improvements, much like asynchronous value iteration with samples in place of expecta-

tions. Actor-critic methods (Chapter 11) maintain a critic, which evaluates, and an actor, which improves. PPO (Chapter 12) improves the policy by a few gradient steps against a learned value function. What changes from chapter to chapter is how evaluation and improvement are approximated: by samples instead of expectations (Part II), and by function approximators instead of tables (Part III). The guarantees of this chapter then hold only approximately, and Section 4.7 shows how much approximation errors can cost.

KEY IDEA

Evaluation and improvement, repeated in any interleaving, converge to an optimal policy: this is generalized policy iteration. It is the common skeleton of dynamic programming and of almost every reinforcement-learning algorithm.

LAB

Lab 4.1: Dynamic programming from scratch (`labs/ch04_dp.py`, about 45 minutes). Run `python3 labs/ch04_dp.py`. You will implement iterative policy evaluation, policy iteration and value iteration in a few lines each and check them against `rlcourse.mdp`, reproduce Table 4.1 and Table 4.3, compare synchronous and in-place sweeps, measure how the number of sweeps grows with γ, try modified policy iteration, and solve a bigger problem, Sutton and Barto's gambler's problem.

TRY IT INTERACTIVELY

Module 3 of the interactive course (`Apps_claude/RL Course/index.html`) runs value-iteration sweeps on the Trail from different starting guesses (all zeros, all tens, random), shows the greedy arrows after each sweep, and lets you improve a policy one step at a time.

4.7 How good is a greedy policy? ●●

In practice the value functions we act on are never exact. Value iteration stops after finitely many sweeps; learning methods estimate values from noisy samples; function approximators cannot represent V^* exactly. Every such method eventually acts greedily with respect to an imperfect value function. How much can that cost?

Lemma 4.9 The loss of a greedy policy

Let V be any value function and π a deterministic policy that is greedy with respect to V. Then

$$\|V^* - V^\pi\|_\infty \le \frac{2\gamma}{1-\gamma}\,\|V - V^*\|_\infty.$$

Proof. Let $\epsilon = \|V - V^*\|_\infty$. Greediness means $\mathcal{T}^\pi V = \mathcal{T}^* V$, so

$$V^* - V^\pi = \big(\mathcal{T}^* V^* - \mathcal{T}^* V\big) + \big(\mathcal{T}^\pi V - \mathcal{T}^\pi V^*\big) + \big(\mathcal{T}^\pi V^* - \mathcal{T}^\pi V^\pi\big),$$

using $V^* = \mathcal{T}^* V^*$ and $V^\pi = \mathcal{T}^\pi V^\pi$. By Theorem 4.3 the first two terms are at most $\gamma\epsilon$ in every state, and the third is $\gamma P^\pi (V^* - V^\pi)$, at most $\gamma\|V^* - V^\pi\|_\infty$ in every state. Since $V^* - V^\pi \ge 0$,

taking the maximum over states gives $\|V^* - V^\pi\|_\infty \le 2\gamma\epsilon + \gamma\|V^* - V^\pi\|_\infty$, and solving for $\|V^* - V^\pi\|_\infty$ proves the bound. ■

The same holds for action values, without the factor γ, because an error in Q already includes the immediate reward.

Corollary 4.10 The loss of a greedy policy, action-value version

If π is greedy with respect to an action-value function Q, that is $\pi(s) \in \arg\max_a Q(s,a)$, then $\|V^* - V^\pi\|_\infty \le \frac{2}{1-\gamma}\|Q - Q^*\|_\infty$.

Proof. Let $\epsilon = \|Q - Q^*\|_\infty$, and in a state s let $a = \pi(s)$ and a^* an optimal action. Then $Q^*(s,a) \ge Q(s,a) - \epsilon \ge Q(s,a^*) - \epsilon \ge Q^*(s,a^*) - 2\epsilon = V^*(s) - 2\epsilon$. So $(\mathcal{T}^\pi V^*)(s) = Q^*(s,\pi(s)) \ge V^*(s) - 2\epsilon$ in every state, and $V^* - V^\pi = (V^* - \mathcal{T}^\pi V^*) + \gamma P^\pi(V^* - V^\pi) \le 2\epsilon + \gamma\|V^* - V^\pi\|_\infty$ in every state. Solve as before. ■

These bounds go back to Singh and Yee (1994) and Williams and Baird (1993). They cannot be improved in general, as the following example shows.

Example 4.4 A value function that misleads as much as possible

Take $\gamma = 0.9$ and two states. State s_1 is absorbing and earns 1 per step, so $V^*(s_1) = 10$. In state s_0 the agent can go to s_1 (reward 0) or stay in s_0 (reward 0.72). Going is optimal: $V^*(s_0) = 0.9 \times 10 = 9$, while staying forever earns only $0.72/0.1 = 7.2$. Now perturb V^* by $\epsilon = 0.1$ in the most damaging direction, $V = (9.1, 9.9)$: slightly too optimistic about staying, slightly too pessimistic about s_1. Under V the two actions in s_0 tie,

$$0.72 + 0.9 \times 9.1 = 8.91 = 0 + 0.9 \times 9.9,$$

and a greedy policy that breaks the tie toward stay loses $9 - 7.2 = 1.8 = \frac{2\times 0.9}{1-0.9} \times 0.1$, exactly the bound. (With a reward slightly above 0.72 for staying, the greedy choice is strict and the loss approaches 1.8.)

♦

The example shows where the factors come from. The error ϵ in the value of each of the two candidate next states shifts the comparison between the actions by up to $2\gamma\epsilon$, so a wrong action can look best whenever its true disadvantage is at most $2\gamma\epsilon$. And the wrong choice is repeated at every visit: losing $2\gamma\epsilon$ per step over the effective horizon $1/(1-\gamma)$ adds up to $2\gamma\epsilon/(1-\gamma)$. With $\gamma = 0.99$, a value function accurate to 0.01 can yield a policy that is worse by almost 2. Accurate values are not enough when the horizon is long; the *ordering* of actions is what matters, and it is most fragile where actions are nearly tied.

Approximate dynamic programming. What if value iteration or policy iteration is run with an error at every step, as happens when each sweep is replaced by fitting a function approximator to sampled backups (Algorithm 9.6)? The errors could accumulate. The following theorem says they do not blow up, but they are amplified.

Theorem 4.11 Error bounds for approximate value and policy iteration

Let $\epsilon \geq 0$, and let π_k denote a policy that is greedy with respect to V_k (in part (a)) or the policy of iteration k (in part (b)).

(a) *Approximate value iteration.* If $\|V_{k+1} - \mathcal{T}^* V_k\|_\infty \leq \epsilon$ for every k, then

$$\limsup_{k\to\infty} \|V_k - V^*\|_\infty \leq \frac{\epsilon}{1-\gamma} \qquad \text{and} \qquad \limsup_{k\to\infty} \|V^* - V^{\pi_k}\|_\infty \leq \frac{2\gamma\epsilon}{(1-\gamma)^2}.$$

(b) *Approximate policy iteration.* If π_{k+1} is greedy with respect to V_k and every evaluation satisfies $\|V_k - V^{\pi_k}\|_\infty \leq \epsilon$, then

$$\limsup_{k\to\infty} \|V^* - V^{\pi_k}\|_\infty \leq \frac{2\gamma\epsilon}{(1-\gamma)^2}.$$

Proof. (a) By the contraction property, $\|V_{k+1} - V^*\|_\infty \leq \|V_{k+1} - \mathcal{T}^* V_k\|_\infty + \|\mathcal{T}^* V_k - \mathcal{T}^* V^*\|_\infty \leq \epsilon + \gamma\|V_k - V^*\|_\infty$. Unrolling, $\|V_k - V^*\|_\infty \leq \gamma^k \|V_0 - V^*\|_\infty + \epsilon(1+\gamma+\cdots+\gamma^{k-1})$, whose limit superior is at most $\epsilon/(1-\gamma)$. Lemma 4.9 turns this into the bound on the greedy policies.

(b) *Step 1: each step loses at most a little.* Using $V^{\pi_k} \geq V_k - \epsilon 1$, monotonicity and the constant shift (Proposition 4.1), greediness of π_{k+1}, and $V_k \geq V^{\pi_k} - \epsilon 1$:

$$\mathcal{T}^{\pi_{k+1}} V^{\pi_k} \geq \mathcal{T}^{\pi_{k+1}} V_k - \gamma\epsilon 1 = \mathcal{T}^* V_k - \gamma\epsilon 1 \geq \mathcal{T}^{\pi_k} V_k - \gamma\epsilon 1 \geq \mathcal{T}^{\pi_k} V^{\pi_k} - 2\gamma\epsilon 1 = V^{\pi_k} - 2\gamma\epsilon 1.$$

As in the proof of Theorem 4.6, $V^{\pi_{k+1}} - V^{\pi_k} = (I - \gamma P^{\pi_{k+1}})^{-1}(\mathcal{T}^{\pi_{k+1}} V^{\pi_k} - V^{\pi_k})$, and the inverse has nonnegative entries with row sums $1/(1-\gamma)$; hence $V^{\pi_{k+1}} \geq V^{\pi_k} - \frac{2\gamma\epsilon}{1-\gamma} 1$.

Step 2: the loss contracts up to a constant. Let $\ell_k = V^* - V^{\pi_k} \geq 0$ and let π^* be optimal. Then

$$\ell_{k+1} = \big(\mathcal{T}^{\pi^*} V^* - \mathcal{T}^{\pi^*} V^{\pi_k}\big) + \big(\mathcal{T}^{\pi^*} V^{\pi_k} - \mathcal{T}^{\pi_{k+1}} V^{\pi_k}\big) + \big(\mathcal{T}^{\pi_{k+1}} V^{\pi_k} - \mathcal{T}^{\pi_{k+1}} V^{\pi_{k+1}}\big).$$

The first term is $\gamma P^{\pi^*} \ell_k$. The second is at most $(\mathcal{T}^{\pi^*} V_k + \gamma\epsilon 1) - (\mathcal{T}^{\pi_{k+1}} V_k - \gamma\epsilon 1) \leq 2\gamma\epsilon 1$, because $\mathcal{T}^{\pi_{k+1}} V_k = \mathcal{T}^* V_k \geq \mathcal{T}^{\pi^*} V_k$. The third is $\gamma P^{\pi_{k+1}}(V^{\pi_k} - V^{\pi_{k+1}}) \leq \frac{2\gamma^2\epsilon}{1-\gamma} 1$ by Step 1. Adding up, $\ell_{k+1} \leq \gamma P^{\pi^*} \ell_k + \frac{2\gamma\epsilon}{1-\gamma} 1$, and since $\ell_k \geq 0$, taking maxima gives $\|\ell_{k+1}\|_\infty \leq \gamma\|\ell_k\|_\infty + \frac{2\gamma\epsilon}{1-\gamma}$. Unrolling as in (a), $\limsup_k \|\ell_k\|_\infty \leq \frac{2\gamma\epsilon}{(1-\gamma)^2}$. ∎

Part (b) is due to Bertsekas and Tsitsiklis (1996), who also allow an error in the greedy step (Exercise 4.14). Two features of the theorem are worth remembering. First, the factor is $1/(1-\gamma)^2$, one power more than for a single greedy step: errors made at different iterations compound. With $\gamma = 0.99$ the constant $2\gamma/(1-\gamma)^2$ is about 20,000, so the bound is mostly a qualitative statement: errors do not explode, and the outcome is as good as the evaluations are accurate. Second, the conclusion is about the limit superior: approximate policy iteration need not converge, and in practice the policies often keep changing among a set of near-optimal ones. Sharper results replace the worst-case maximum norm by weighted norms under the state distributions actually encountered, at the price of *concentrability coefficients* that measure how these distributions differ (Munos 2003, 2007); Chapter 9 and Chapter 17 return to them.

4.8 Convergence rates and complexity ●●

How much computation does dynamic programming need? Suppose rewards lie in $[0, r_{\max}]$ and value iteration starts from $V_0 = 0$, so that $\|V_0 - V^*\|_\infty \le r_{\max}/(1-\gamma)$. By Corollary 4.5, $\|V_k - V^*\|_\infty \le \varepsilon$ as soon as $\gamma^k r_{\max}/(1-\gamma) \le \varepsilon$, that is, after

$$k \ge \frac{\log\big(r_{\max}/(\varepsilon(1-\gamma))\big)}{\log(1/\gamma)} \quad \text{sweeps; since } \log(1/\gamma) \ge 1-\gamma, \quad k = \left\lceil \frac{1}{1-\gamma} \log \frac{r_{\max}}{\varepsilon(1-\gamma)} \right\rceil \text{ suffices.} \tag{4.4}$$

The number of sweeps is $O\big(\log(1/(\varepsilon(1-\gamma)))/(1-\gamma)\big)$ for $r_{\max} = 1$: logarithmic in the accuracy but linear in the effective horizon. For $r_{\max} = 1$ and $\varepsilon = 0.01$, the exact bound is 66 sweeps when $\gamma = 0.9$ and 917 when $\gamma = 0.99$.

Each sweep computes $|\mathcal{S}||\mathcal{A}|$ backups, and a backup sums over the possible next states, so a sweep costs $O(|\mathcal{S}|^2|\mathcal{A}|)$ operations in general and $O(|\mathcal{S}||\mathcal{A}|b)$ if each state–action pair has at most b possible successors. Policy iteration needs $O(|\mathcal{S}|^3)$ operations per exact evaluation (a linear solve) plus $O(|\mathcal{S}|^2|\mathcal{A}|)$ per improvement. Its number of iterations is bounded by $|\mathcal{A}|^{|\mathcal{S}|}$ (Corollary 4.7) and also by the number of value-iteration sweeps, since $\|V^* - V^{\pi_{k+1}}\|_\infty \le \gamma \|V^* - V^{\pi_k}\|_\infty$ (Exercise 4.10). In practice it is much smaller than either bound. Policy iteration can be interpreted as Newton's method applied to the equation $\mathcal{T}^* V - V = 0$, which explains its fast convergence near the solution (Puterman 1994).

Figure 4.6 shows these rates in experiments. On random continuing MDPs the number of value-iteration sweeps grows in proportion to $1/(1-\gamma)$ and follows the bound closely, while policy iteration needs three or four iterations whatever γ is. On the episodic 4×3 world, value iteration's sweep count hardly grows with γ: good policies reach an exit within a few steps, so the far future, which is what makes large γ expensive, contributes little.

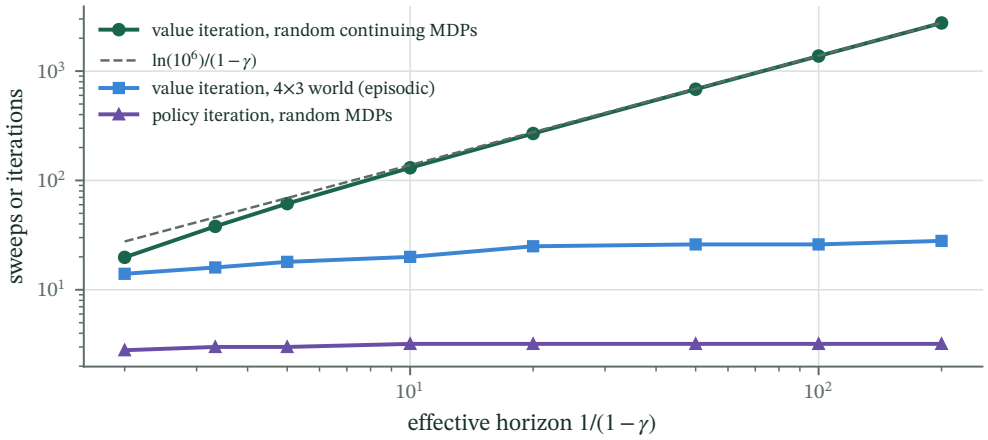


Figure 4.6. Sweeps of value iteration until $\|V_k - V^*\|_\infty \le 10^{-6}\|V^*\|_\infty$ (from $V_0 = 0$), against the effective horizon $1/(1-\gamma)$, on log–log axes: averages over five random continuing MDPs with 20 states and 4 actions, and the episodic 4×3 world. The dashed line is the worst-case rate $\ln(10^6)/(1-\gamma)$. Policy iteration (triangles) needs three or four iterations on the same random MDPs for every γ.

The curse of dimensionality. All of these costs are polynomial in the number of states. The trouble is that the number of states is usually exponential in the size of the problem's description:

a system described by n variables with ten values each has 10^n states. Bellman called this the *curse of dimensionality* (Bellman 1957b). It is the reason dynamic programming in its exact, tabular form is limited to problems with at most millions or perhaps billions of states, and the reason Part III of this book replaces tables by function approximators.

4.8.1 Strongly polynomial bounds ●●●

An algorithm is *strongly polynomial* if its number of arithmetic operations is bounded by a polynomial in the number of states and actions alone, independently of the numerical values of the transition probabilities and rewards. Value iteration is not strongly polynomial. The number of sweeps it needs before its greedy policy is optimal depends on how close the best and second-best actions are: if two actions in some state differ in value by a tiny amount, the values must be computed very precisely before the greedy choice settles, and that takes about $\log(1/\text{gap})/(1-\gamma)$ sweeps.

Policy iteration is different. For a *fixed* discount factor, Ye (2011) proved that Howard's policy iteration, and the simplex method with Dantzig's pivoting rule applied to the linear program of Section 4.9, terminate after a number of iterations bounded by a polynomial in $|\mathcal{S}|$ and $|\mathcal{A}|$ (with a factor depending on $1/(1-\gamma)$): both are strongly polynomial. Hansen et al. (2013) improved the bound for policy iteration and extended it to two-player turn-based stochastic games, and Scherrer (2016) showed that Howard's policy iteration needs at most $O\big(\frac{m}{1-\gamma}\log\frac{1}{1-\gamma}\big)$ iterations, where $m = |\mathcal{S}||\mathcal{A}|$ is the number of state–action pairs. When the discount factor is not fixed, the picture changes: Fearnley (2010) constructed MDPs, under the total-reward and average-reward criteria, on which Howard's policy iteration needs exponentially many iterations. Whether MDPs can be solved in strongly polynomial time when γ is part of the input is open, as is the corresponding question for linear programming in general. From the point of view of parallel computation, solving MDPs is hard in a different sense: Papadimitriou and Tsitsiklis (1987) showed that it is P-complete, so it is unlikely to admit dramatic speed-ups from parallel processors.

4.9 Linear programming ●●●

Section 3.10.3 wrote the MDP as a pair of linear programs: the primal, which minimizes $\sum_s w(s)V(s)$ over value functions subject to $V(s) \geq r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')$ for all s, a, and the dual, which maximizes $\frac{1}{1-\gamma}\sum_{s,a} d(s,a)r(s,a)$ over occupancy measures $d \geq 0$ subject to the flow constraints $\sum_a d(s',a) = (1-\gamma)w(s') + \gamma \sum_{s,a} P(s' \mid s,a)d(s,a)$. With the tools of this chapter we can prove that they solve the MDP, and see how the dual reveals an optimal policy.

Proposition 4.12 Linear programs for an MDP

Let the weights satisfy $w(s) > 0$ and $\sum_s w(s) = 1$.

(a) Every feasible V of the primal LP satisfies $V \geq V^*$; hence V^* is its unique optimal solution.

(b) For every feasible V of the primal and every feasible d of the dual,

$$\sum_s w(s)V(s) - \frac{1}{1-\gamma}\sum_{s,a} d(s,a)\, r(s,a)$$
$$= \frac{1}{1-\gamma}\sum_{s,a} d(s,a)\Big[V(s) - r(s,a) - \gamma\sum_{s'} P(s' \mid s,a)\, V(s')\Big] \geq 0.$$

(c) The occupancy measure of an optimal policy, started from w, is an optimal solution of the dual, and both programs have the optimal value $\sum_s w(s)V^*(s)$. Every optimal dual solution d puts weight only on optimal actions: $d(s,a) > 0$ implies $Q^*(s,a) = V^*(s)$.

Proof. (a) Feasibility says $V \geq \mathcal{T}^*V$. By monotonicity (Proposition 4.1), $V \geq \mathcal{T}^*V \geq (\mathcal{T}^*)^2V \geq \cdots$, and $(\mathcal{T}^*)^kV \to V^*$ by Corollary 4.5, so $V \geq V^*$. Since $V^* = \mathcal{T}^*V^*$, V^* is feasible, and because all weights are positive, any other feasible V has a strictly larger objective.

(b) Write $\mu(s) = \sum_a d(s,a)$. The dual constraint says $\gamma\sum_{s,a} P(s' \mid s,a)d(s,a) = \mu(s') - (1-\gamma)w(s')$. Hence

$$\sum_{s,a} d(s,a)\Big[V(s) - \gamma\sum_{s'} P(s' \mid s,a)V(s')\Big] = \sum_s \mu(s)V(s) - \sum_{s'}\big(\mu(s') - (1-\gamma)w(s')\big)V(s')$$
$$= (1-\gamma)\sum_s w(s)V(s),$$

which rearranges to the identity. Every bracket is nonnegative by primal feasibility and every $d(s,a)$ is nonnegative, so the sum is nonnegative.

(c) Occupancy measures satisfy the flow constraints (Section 3.10.3), so the occupancy measure d^{π^*} of an optimal policy is feasible, and by the occupancy formula $J(\pi) = \frac{1}{1-\gamma}\sum_{s,a} d^\pi(s,a)r(s,a)$ of Chapter 3 its objective equals $\sum_s w(s)V^{\pi^*}(s) = \sum_s w(s)V^*(s)$. So the gap in (b) between V^* and d^{π^*} is zero, and by (b) neither can be improved: both are optimal, with equal values. For any optimal d the gap with V^* is also zero, so every term vanishes; the bracket for V^* equals $V^*(s) - Q^*(s,a)$, so $d(s,a) > 0$ forces $Q^*(s,a) = V^*(s)$.

■

Part (c) is *complementary slackness*: a positive dual variable means a tight primal constraint, and a tight constraint means an optimal action. So an optimal dual solution is an optimal policy in disguise: $\pi(a \mid s) = d(s,a)/\sum_{a'} d(s,a')$. On the Trail with $\gamma = 0.9$ and all the weight on Start (the dual makes sense for any nonnegative weights), an optimal dual solution is $d(\text{Start}, \text{go}) = 0.1$, $d(\text{Middle}, \text{go}) = 0.09$, $d(\text{Goal}, \text{finish}) = 0.081$ and 0.729 on End, which is the occupancy measure of the policy that goes all the way, with objective $10 \times 0.081 \times 4 = 3.24 = V^*(\text{Start})$. The exit constraints are slack by $3.24 - 1 = 2.24$ in Start and $3.6 - 2 = 1.6$ in Middle, the amounts by which exiting falls short.

Linear programs can be solved in time polynomial in the size of their description, so this gives a polynomial-time algorithm for finite MDPs, and simplex-type methods on the dual are closely related to policy iteration: a pivot changes the action in one state (Ye 2011). For large unconstrained problems, dynamic programming is usually faster in practice. The LP view earns its

keep elsewhere: in constrained MDPs, which add linear constraints on occupancy measures (Section 3.10.3, Chapter 23); in approximate linear programming, which restricts V to a linear combination of features (de Farias and Van Roy 2003); and in the duality-based estimators of offline reinforcement learning (Chapter 17).

4.10 Finite horizons and beyond tables

Backward induction. In a problem with a fixed horizon of H decisions (Chapter 3), dynamic programming is even simpler: there is nothing to converge. Starting from $V_H^* = 0$ (or from given terminal values), compute $V_{H-1}^*, \dots, V_0^*$ backward with $V_t^*(s) = \max_a \big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V_{t+1}^*(s') \big]$, recording at each stage t a greedy action $\pi_t^*(s)$. After exactly H sweeps, at a cost of H times that of one sweep, the time-dependent policy $(\pi_0^*, \dots, \pi_{H-1}^*)$ is optimal among all policies for the H-step problem; the proof is the argument of Step 2 in Theorem 3.7, with a finite sum (Puterman 1994, ch. 4).

Backward induction and value iteration are the same computation viewed differently: starting from $V_0 = 0$, the k-th iterate of value iteration is the optimal value of the problem with k decisions left. This gives the promised bound on how much a finite horizon matters: $V_k = (\mathcal{T}^*)^k 0$, so by Corollary 4.5 the H-step optimal values differ from the infinite-horizon V^* by at most $\gamma^H \|V^*\|_\infty \le \gamma^H r_{\max}/(1-\gamma)$.

Continuous states. The Bellman equations make sense for continuous state spaces too, but a value function is then a function on a continuum, and a sweep cannot update "every state". Discretizing the state space runs straight into the curse of dimensionality; approximating the value function with a parametric family leads to approximate dynamic programming and to Part III. There is one important case in which dynamic programming over a continuous state space can be done exactly, by hand.

4.10.1 The linear-quadratic regulator ●●

In the *linear-quadratic regulator* (LQR), the state $s_t \in \mathbb{R}^n$ and the action $a_t \in \mathbb{R}^m$ are vectors, the dynamics are linear and the cost is quadratic:

$$s_{t+1} = A s_t + B a_t, \qquad c(s,a) = s^\top M s + a^\top N a,$$

with M symmetric positive semidefinite and N symmetric positive definite. The goal is to minimize the discounted cost $\sum_t \gamma^t c(s_t, a_t)$, which is the same as maximizing the reward $-c$. (Control texts usually write x and u for the state and action and Q and R for the cost matrices; we rename them to avoid a clash with action values and rewards.) Typical uses are keeping a system near an operating point: a drone hovering, a vehicle following a lane, a chemical process held at a set point.

Backward induction in closed form. Suppose the optimal cost with $k-1$ steps left is a quadratic function $C_{k-1}(s) = s^\top K_{k-1} s$ with K_{k-1} symmetric positive semidefinite; with no steps left, $K_0 = 0$. With k steps left the agent minimizes over a

$$s^\top M s + a^\top N a + \gamma\, (As + Ba)^\top K_{k-1} (As + Ba).$$

This is a convex quadratic function of a (its Hessian $2(N + \gamma B^\top K_{k-1} B)$ is positive definite), so setting its gradient $2Na + 2\gamma B^\top K_{k-1}(As + Ba)$ to zero gives the unique minimizer $a = -L_k s$, a *linear* function of the state, with gain

$$L_k = \gamma \left(N + \gamma B^\top K_{k-1} B\right)^{-1} B^\top K_{k-1} A.$$

Substituting it back shows that the optimal cost with k steps left is again quadratic, $C_k(s) = s^\top K_k s$, with

$$K_k = M + \gamma A^\top K_{k-1} A - \gamma A^\top K_{k-1} B\, L_k. \tag{4.5}$$

This is the *Riccati recursion*. By induction, every stage of backward induction keeps the quadratic form, and dynamic programming over $\mathbb{R}^n$ reduces to a recursion on $n \times n$ matrices.

Infinite horizon. As $k \to \infty$ the matrices K_k increase (more steps cannot cost less, because costs are nonnegative), and if some linear policy has finite discounted cost they are bounded, so they converge to a matrix K that solves the *algebraic Riccati equation*, the fixed-point version of (4.5). Under standard conditions, for example M positive definite, the stationary linear policy $a = -Ls$ with the limiting gain is then optimal for the infinite-horizon problem (Bertsekas 2017, 2012). Zero-mean noise added to the dynamics changes the cost but not the optimal gains (Exercise 4.13).

Example 4.5 Stabilizing an unstable scalar system

Take $n = m = 1$, $s_{t+1} = 1.2\, s_t + a_t$, cost $s^2 + 0.5\, a^2$ and $\gamma = 0.95$. Left alone, the state grows by 20% per step. In the scalar case the recursion reads $K_k = M + \gamma A^2 K_{k-1} N/(N + \gamma B^2 K_{k-1})$, and from $K_0 = 0$ it gives

$$K_1 = 1, \qquad K_2 = 1 + \frac{0.95 \times 1.44 \times 0.5}{0.5 + 0.95} \approx 1.4717, \qquad K_3 \approx 1.5303,$$

then $K_4 \approx 1.5357$, and the sequence converges to $K \approx 1.5363$. The limiting gain is $L = 0.95 \times 1.2 \times K/(0.5 + 0.95\, K) \approx 0.894$, so the optimal policy is $a = -0.894\, s$ and the controlled system evolves as $s_{t+1} = (1.2 - 0.894)\, s_t \approx 0.306\, s_t$: it now shrinks by about 70% per step. The optimal discounted cost from state s is about $1.536\, s^2$.

♦

The LQR is the starting point of much of modern control: iterative LQR and model-predictive control apply it repeatedly to local linear models of nonlinear systems (Chapter 15), and it is the cleanest test case for the theory of policy gradients (Chapter 14).

HISTORICAL NOTE

Bellman developed dynamic programming at the RAND Corporation in the early 1950s and collected it in his 1957 book (Bellman 1957b), which also introduced value iteration and the phrase "curse of dimensionality". Ronald Howard's policy iteration followed in 1960 (Howard 1960). The contraction-mapping view that organizes this chapter was made systematic by Denardo (1967), building on the fixed-point theorem of Banach (1922). Modified policy iteration is due to Puterman and Shin (1978), and the linear-programming formulation to Manne (1960).

4.11 Summary

- Dynamic programming computes value functions and optimal policies of a known MDP by using the Bellman equations as update rules; one update of every state is a sweep.
- *Iterative policy evaluation* repeats $V \leftarrow \mathcal{T}^{\pi}V$. On the Trail it is exact after three sweeps; on the slippery Trail its error shrinks geometrically. In-place sweeps use new values at once and, in a good order, converge much faster.
- The Bellman operators are *monotone* and are γ*-contractions* in the maximum norm. By Banach's fixed-point theorem each has exactly one fixed point (V^{π} or V^{*}), iteration from any start converges to it with error at most γ^{k} times the initial error, and a last change of θ certifies an error of at most $\gamma\theta/(1-\gamma)$.
- The *policy improvement theorem*: a policy that is greedy with respect to V^{π} is at least as good as π in every state, and strictly better somewhere unless π is optimal. *Policy iteration* alternates exact evaluation and greedy improvement and stops after finitely many iterations, usually very few, with an optimal policy.
- *Value iteration* repeats $V \leftarrow \mathcal{T}^{*}V$. Stopping when the last change is below $\varepsilon(1-\gamma)/(2\gamma)$ guarantees an ε-optimal greedy policy; the greedy policy is often optimal long before the values converge.
- *Generalized policy iteration*, any interleaving of evaluation and improvement, contains policy iteration, value iteration, modified policy iteration and asynchronous DP, and it is the skeleton of nearly every reinforcement-learning algorithm.
- (●●) Acting greedily on a value function with error ϵ loses at most $2\gamma\epsilon/(1-\gamma)$, and this is tight; with errors at every iteration, approximate value and policy iteration lose at most $2\gamma\epsilon/(1-\gamma)^2$ in the limit.
- (●●) Value iteration needs $O(\log(1/(\varepsilon(1-\gamma)))/(1-\gamma))$ sweeps of cost $O(|\mathcal{S}|^2|\mathcal{A}|)$; the number of states itself grows exponentially with the number of state variables, the curse of dimensionality. (●●●) For a fixed γ, policy iteration is strongly polynomial; MDPs can also be solved by linear programming, whose dual over occupancy measures exposes an optimal policy through complementary slackness.
- Finite-horizon problems are solved exactly by backward induction, and (●●) the linear-quadratic regulator is solved by backward induction in closed form, through the Riccati recursion.

The next chapter drops the assumption that the model is known. Monte Carlo methods evaluate and improve policies from sampled episodes alone, replacing the expectations of dynamic programming by averages of returns.

4.12 Further Reading

Sutton and Barto (2018, ch. 4) introduce dynamic programming with the gambler's problem and a car-rental example and coined the term generalized policy iteration. Puterman (1994, ch. 6) is the reference for value iteration, policy iteration, modified policy iteration, Gauss–Seidel variants and the linear program, with complete proofs; Bertsekas (2012) and Bertsekas (2017) cover the same ground from the control viewpoint, including stochastic shortest-path problems

and the linear-quadratic regulator, and Bertsekas and Tsitsiklis (1996) develop asynchronous and approximate dynamic programming and the error bounds of Section 4.7. Szepesvári (2010) gives a compact modern summary.

The original papers remain worth reading: Bellman (1957b), Howard (1960), and Denardo (1967) for the contraction view. The bounds on greedy policies are in Singh and Yee (1994) and Williams and Baird (1993); Munos (2003) and Munos (2007) refine the approximate-DP bounds with weighted norms, which is the form in which they are used in modern analyses of fitted value iteration (Chapter 9).

(●●●) On complexity, Littman et al. (1995) survey what was known about solving MDPs, Papadimitriou and Tsitsiklis (1987) prove P-completeness, and Ye (2011), Hansen et al. (2013) and Scherrer (2016) establish and sharpen the strongly polynomial bounds for policy iteration with a fixed discount factor, while Fearnley (2010) shows exponential behavior without one. For approximate linear programming with features, see de Farias and Van Roy (2003).

4.13 Exercises

Exercise 4.1 ● Sweeps by hand

On the Trail with $\gamma = 0.9$, evaluate the policy with $p_0 = 0.8$ and $p_1 = 0.5$. (a) Run synchronous sweeps from $V_0 = 0$ until nothing changes, and compare the result with the closed-form values of Section 3.6. (b) Repeat from the optimistic guess $V_0 = 10$ in Start, Middle and Goal (and 0 in End). (c) How many in-place sweeps are needed in the order Goal, Middle, Start?

Exercise 4.2 ● Value iteration on the slippery Trail

On the slippery Trail with $\gamma = 0.9$, run value iteration from $V_0 = 0$. (a) Compute V_1, V_2 and V_3 by hand and check them against Table 4.3. (b) Find the greedy policy after each of these sweeps. (c) With $\varepsilon = 0.01$, after how many sweeps does the stopping rule of Algorithm 4.3 fire, and how large is the true error $\|V_k - V^*\|_\infty$ then? (Use the lab.)

Exercise 4.3 ● Policy iteration on the slippery Trail

Run policy iteration on the slippery Trail, starting from the policy that exits in both Start and Middle: (a) with $\gamma = 0.9$; (b) with $\gamma = 0.5$. In each case list the evaluations and improvement steps, and relate the outcome to the threshold $\gamma = 5/9$ of Section 3.7.

Exercise 4.4 ● How many sweeps?

Rewards lie in $[0, 1]$ and value iteration starts from $V_0 = 0$. (a) How many sweeps guarantee $\|V_k - V^*\|_\infty \le 0.01$ when $\gamma = 0.9$? When $\gamma = 0.99$? (b) For each γ, what threshold on $\|V_k - V_{k-1}\|_\infty$ guarantees that the greedy policy is 0.01-optimal? (c) Explain in one paragraph why the number of sweeps grows like $1/(1-\gamma)$, and why Figure 4.6 shows no such growth for the 4×3 world.

Exercise 4.5 • The order of in-place sweeps

The order of in-place updates matters. (a) Explain why in-place evaluation of any policy on an MDP without cycles (like the Trail) is exact after one sweep if every state is updated after all the states that can follow it. (b) For the 4×3 world with $\gamma = 0.9$, propose an order for in-place value iteration that should converge quickly, and explain why. (c) Using the lab, count the sweeps needed to reach $\|V - V^*\|_\infty \le 10^{-6}$ for synchronous sweeps, for in-place sweeps row by row, and for your order.

Exercise 4.6 • Choosing the amount of evaluation

Run modified policy iteration on the 4×3 world ($\gamma = 0.9$, target accuracy 10^{-6}) for $m = 1, 2, 3, 5, 10, 20, 50$ evaluation sweeps per improvement, using the function `modified_policy_iteration` of `rlcourse.dp_examples`. Count the work, charging 4 units for an improvement (it compares four actions) and 1 unit for each evaluation sweep. Which m is cheapest? Explain the shape of the curve.

Exercise 4.7 • Planning at scale

An MDP has 10^6 states, 4 actions, and at most 10 possible next states for each state–action pair. (a) Estimate the number of arithmetic operations in one sweep of value iteration, and the memory needed to store the model. (b) Why is exact policy evaluation by a dense linear solve out of the question, and what would you do instead? (c) Recommend an algorithm for $\gamma = 0.99$ and justify it with the costs of Section 4.6 and Section 4.8.

Exercise 4.8 •• Monotone value iteration

Monotonicity gives more than convergence. (a) Prove Proposition 4.1 in detail. (b) Show that if $V_0 \le \mathcal{T}^* V_0$, then value iteration produces a nondecreasing sequence $V_0 \le V_1 \le V_2 \le \cdots$, and that if $V_0 \ge \mathcal{T}^* V_0$ it is nonincreasing. (c) Deduce that from $V_0 = 0$ value iteration increases monotonically to V^* whenever all rewards are nonnegative, as on the Trail, and give a starting point from which it decreases monotonically.

Exercise 4.9 •• Which norm?

The choice of norm in Theorem 4.3 is essential. (a) Prove Lemma 4.2 and the corresponding inequality for minima. (b) Let $P^\pi = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ and $\gamma = 0.9$. Find vectors U, W with $\|\mathcal{T}^\pi U - \mathcal{T}^\pi W\|_2 > \|U - W\|_2$, so that $\mathcal{T}^\pi$ is not a contraction in the Euclidean norm. (c) Let μ be a stationary distribution of P^π, that is $\mu^\top P^\pi = \mu^\top$, and define $\|x\|_\mu^2 = \sum_s \mu(s) x(s)^2$. Show that $\|P^\pi x\|_\mu \le \|x\|_\mu$, so $\mathcal{T}^\pi$ is a γ-contraction in $\|\cdot\|_\mu$. (Chapter 9 builds on this fact.)

Exercise 4.10 •• Policy iteration is at least as fast as value iteration

Let π_{k+1} be greedy with respect to V^{π_k}. (a) Show that $V^{\pi_{k+1}} \geq \mathcal{T}^* V^{\pi_k} \geq V^{\pi_k}$. (b) Deduce $\|V^* - V^{\pi_{k+1}}\|_\infty \leq \gamma \|V^* - V^{\pi_k}\|_\infty$, so that policy iteration needs no more iterations than value iteration started from V^{π_0} needs sweeps to reach the same accuracy.

Exercise 4.11 ●● Policy iteration that never stops

On the Trail with $\gamma = 0.5$, every action ties in both Start and Middle (Chapter 1). (a) Run policy iteration with the rule "among the maximizing actions, choose one different from the current action whenever possible". Show that it never stops. (b) Show that every policy it visits is optimal, so the failure is only in the stopping test. (c) Give two ways to repair the algorithm and prove that each terminates.

Exercise 4.12 ●● The greedy-policy bound is tight

This exercise studies Example 4.4 further. (a) Verify its numbers. (b) Change the reward for staying to $0.72 + \delta$ with $0 < \delta < 0.18$. Show that the same V is still within 0.1 of V^*, that the greedy policy now strictly prefers to stay, and compute its loss as a function of δ. (c) Construct a similar example in which a greedy policy for an action-value function Q with $\|Q - Q^*\|_\infty = \epsilon$ loses exactly $2\epsilon/(1-\gamma)$, showing that Corollary 4.10 is tight too.

Exercise 4.13 ●● A scalar Riccati equation

Consider $s_{t+1} = 0.9\, s_t + 0.5\, a_t$ with cost $s^2 + a^2$ and no discounting ($\gamma = 1$). (a) Compute K_1, K_2, K_3 from the Riccati recursion (4.5) with $K_0 = 0$. (b) Find the limit K by solving the algebraic Riccati equation, the optimal gain L and the closed-loop factor $0.9 - 0.5L$. (c) Return to Example 4.5 and add noise: $s_{t+1} = 1.2\, s_t + a_t + w_t$ with $\mathbb{E}[w_t] = 0$ and $\mathbb{E}[w_t^2] = \sigma^2$, independent over time. Show that the optimal gain does not change and that the optimal cost becomes $Ks^2 + \gamma\sigma^2 K/(1-\gamma)$.

Exercise 4.14 ●● Approximate improvement

Extend the proof of Theorem 4.11 to the case where the improvement step is itself approximate: π_{k+1} satisfies $\mathcal{T}^{\pi_{k+1}} V_k \geq \mathcal{T}^* V_k - \delta \mathbf{1}$ for some $\delta \geq 0$. Show that $\limsup_k \|V^* - V^{\pi_k}\|_\infty \leq (\delta + 2\gamma\epsilon)/(1-\gamma)^2$.

Exercise 4.15 ●●● The Trail as a linear program

This exercise solves the Trail by linear programming. (a) Write the dual LP of Section 3.10.3 for the Trail with weights $w = d_0$ (all weight on Start), with one variable per state–action pair. (b) For $\gamma = 0.9$, find an optimal solution by hand, verify it with `rlcourse.mdp_tools.lp_dual`, and check complementary slackness against Q^*. (c) For $\gamma = 0.5$, show that the dual has several optimal solutions, describe all of them, and explain the connection with the ties of Exercise 4.11. (d) Why does the primal LP need strictly positive weights to determine V^* in every state? With $w = d_0$ and $\gamma = 0.4$, where exiting at once is optimal, find a feasible $V \neq V^*$ with the same objective value as V^*.

Exercise 4.16 ●●● Dynamic programming without discounting

Russell and Norvig solve the 4×3 world with $\gamma = 1$ (Section 3.7). (a) Show that the policy "always move left" never reaches an exit from the bottom-left cell, and compute its value. (b) Show that with $\gamma = 1$ the operator $\mathcal{T}^*$ is not a contraction in the maximum norm. (c) Read the discussion of stochastic shortest-path problems in Bertsekas and Tsitsiklis (1996, ch. 2) and explain the conditions under which value iteration still converges, and why the 4×3 world satisfies them. (d) Check numerically that value iteration converges to Russell and Norvig's values from $V_0 = 0$ and from $V_0 = -10$.

Exercise 4.17 ●●● How many iterations does policy iteration need?

This is an experimental and reading exercise. (a) Using `rlcourse.mdp.random_mdp`, measure the number of iterations of policy iteration for random MDPs with $|\mathcal{S}| = 5, 10, 20, 50, 100, 200$ states, 4 actions and $\gamma = 0.9$ and 0.99, averaging over several seeds. (b) Compare with the bound of Scherrer (2016) and comment on the gap. (c) Read the introductions of Ye (2011) and Fearnley (2010) and explain in a paragraph what "strongly polynomial" means, why a fixed discount factor matters, and what is known when it is not fixed. Which open problem would a strongly polynomial algorithm for MDPs with γ in the input bear on?

Part II

Learning from Experience

In Part II the agent no longer knows how its environment works; it must learn from the episodes it lives through. Chapter 5 learns from complete returns, Chapter 6 from one-step predictions of predictions, the central idea of the field, and Chapter 7 spans the range between them. Chapter 8 lets the agent learn a model of its world and plan with it. Every problem here is small enough to store one number per state, which keeps the ideas in plain view.

CHAPTER 5

Monte Carlo Methods

WHAT YOU WILL LEARN

- Explain how averaging the returns of complete episodes estimates V^π and Q^π without a model, and why the estimates are unbiased and consistent.
- Implement first-visit and every-visit Monte Carlo prediction incrementally, with sample-average and constant step sizes, and predict how their errors behave.
- Implement Monte Carlo control with exploring starts and with ε-soft policies, and explain why it needs exploration.
- Evaluate one policy from episodes generated by another with ordinary and weighted importance sampling, and compute both estimates by hand.
- Show that the variance of importance sampling can grow exponentially with the horizon or be infinite, and derive per-decision importance sampling (●●).
- Compare first-visit and every-visit estimators statistically, and compute confidence intervals and sample sizes for Monte Carlo estimates (●●).
- Relate Monte Carlo estimation to regression targets, control variates and tree search (●●●).

Chapters 3 and 4 solved decision problems whose rules were written down. For every state and action we knew the probabilities of the next states and the expected rewards, and dynamic programming turned that knowledge into exact values and optimal policies. Most real problems come without such a description. Nobody can write down the transition probabilities of a game against a human opponent, of a robot arm pushing an object across a table, or of a reader's response to a recommended article. What we can do is act and watch what happens.

This chapter studies the most direct way to learn from what happens: play complete episodes and average the returns that follow each state. Chapter 1 already did this once, when it estimated the value 1.76 of the Trail's uniform policy by averaging the returns of many episodes and watched the error shrink like $1/\sqrt{n}$. Methods of this kind are called *Monte Carlo methods*, after the casino, because they replace exact calculation by random sampling. They need no model. They do not build one estimate on another, so an error in one state's estimate does not spread to the others. And they are easy to analyze, because an average of independent samples is the best-understood object in statistics.

Monte Carlo methods also have two weaknesses, and these motivate the rest of Part II. They learn nothing until an episode ends, so in their basic form they apply only to tasks that come in episodes. And their estimates can be noisy, because a return is a sum of many random rewards. The noise becomes severe when we try to learn about one policy from the data of another, which is the problem of *off-policy* learning. This chapter solves that problem with importance sampling,

and then shows how fragile the solution is.

Section 5.1 estimates values from complete episodes, and Section 5.2 turns the averages into incremental updates with step sizes. Section 5.3 uses the estimates to improve policies, which forces us to think about exploration. Section 5.4 learns about one policy from episodes generated by another. These four sections are the Foundations material and form a complete introduction. The Graduate sections explain why importance sampling can fail (Section 5.5) and study the statistics of Monte Carlo estimates (Section 5.6). The Research section, Section 5.7, connects Monte Carlo estimation to function approximation, variance reduction and tree search. On a first reading, read Sections 5.1 to 5.4 and the summary.

5.1 Learning from complete episodes

Suppose we want the value of a policy π in a problem whose model we do not know. By definition (Definition 3.4), the value of a state s is the expected return when the agent starts in s and follows π:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s], \qquad G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{T-t-1} r_{T-1}.$$

Here r_t is the reward received right after the action a_t in state s_t, $\gamma \in [0, 1]$ is the discount factor, and T is the time step at which the episode reaches a terminal state. An expected value can be estimated by averaging samples. Every time the agent visits s and then follows π to the end of the episode, the return that follows is one sample of a random variable whose mean is $V^\pi(s)$. Averaging many such returns estimates $V^\pi(s)$. That is the whole idea of *Monte Carlo prediction*: *prediction* because it evaluates a given policy, *Monte Carlo* because it averages random samples.

Two practical points make this easy to implement. Returns are best computed backward from the end of the episode, with the recursion

$$G_t = r_t + \gamma\, G_{t+1}, \qquad G_T = 0,$$

which Chapter 3 derived from the definition of the return. And the agent only needs to *generate* episodes, from the real environment or from a simulator. It never needs the transition probabilities $P(s' \mid s, a)$ or the expected rewards $r(s, a)$.

5.1.1 The Trail from sampled episodes

Let us estimate the values of the Trail's uniform policy ($p_0 = p_1 = \frac{1}{2}, \gamma = 0.9$) as an agent would, from six episodes. Table 5.1 lists six episodes produced by the lab's simulator. For each episode it gives the rewards and, for every state the episode visits, the return that follows the visit.

Table 5.1. Six episodes of the uniform policy on the Trail ($\gamma = 0.9$), the returns that follow each visited state, and the Monte Carlo estimates (the averages). The true values are 1.76, 2.8 and 4.

Episode	Path	Rewards	Return from Start	from Middle	from Goal
1	go, exit	0, 2	1.80	2.00	
2	exit	1	1.00		
3	exit	1	1.00		
4	go, go, finish	0, 0, 4	3.24	3.60	4.00
5	go, go, finish	0, 0, 4	3.24	3.60	4.00
6	go, exit	0, 2	1.80	2.00	
Average			2.01	2.80	4.00

Read episode 4 from right to left. The return from Goal is the last reward, 4. The return from Middle is $0 + 0.9 \times 4 = 3.6$, and the return from Start is $0 + 0.9 \times 3.6 = 3.24$. The Monte Carlo estimate of each state's value is the average of the returns in its column:

$$\hat{V}(\text{Start}) = \frac{1.8 + 1 + 1 + 3.24 + 3.24 + 1.8}{6} = \frac{12.08}{6} \approx 2.01,$$
$$\hat{V}(\text{Middle}) = \frac{2 + 3.6 + 3.6 + 2}{4} = 2.8.$$

Three things are worth noticing. The estimate for Goal is exact after one visit, because the return from Goal is always 4: no randomness, no error. The estimate for Middle happens to be exactly right, because two of its four returns were 2 and two were 3.6, exactly the proportions of the policy. The estimate for Start is 0.25 too high, because two of the six episodes went all the way, more than the one in four that the policy produces on average. Finally, each state's estimate uses only the returns observed from that state. Nothing is shared between states, unlike in dynamic programming, where the value of Start is computed from the value of Middle.

5.1.2 First visits and every visit

On the Trail no state is visited twice in an episode. On the *slippery Trail* of Chapter 3, each go fails with probability 0.2 and leaves the hiker where she is, so a state can be visited several times. Here is a possible episode:

$$\text{Start} \xrightarrow{\text{go (slips), 0}} \text{Start} \xrightarrow{\text{go, 0}} \text{Middle} \xrightarrow{\text{exit, +2}} \text{End}.$$

Its returns are $G_2 = 2$, $G_1 = 0 + 0.9 \times 2 = 1.8$ and $G_0 = 0 + 0.9 \times 1.8 = 1.62$. The state Start is visited at $t = 0$ and again at $t = 1$, so this one episode offers two returns for Start: 1.62 and 1.8.

There are two natural choices. *First-visit Monte Carlo* uses, from each episode, only the return that follows the first visit to s (1.62 here). *Every-visit Monte Carlo* uses the returns that follow

all visits (both 1.62 and 1.8, whose average is 1.71). Both converge to $V^\pi(s)$ as the number of episodes grows. They differ for small samples. First-visit Monte Carlo averages independent returns and is unbiased. Every-visit Monte Carlo averages returns that overlap (the return 1.62 contains the return 1.8, discounted) and has a small bias that vanishes as the number of episodes grows. Section 5.6 works out the difference. Both are used in practice: first-visit Monte Carlo is easier to analyze, and every-visit Monte Carlo extends more naturally to function approximation (Chapter 9) and to eligibility traces (Chapter 7).

Algorithm 5.1 First-visit Monte Carlo prediction

Input: the policy π to evaluate; discount γ.

1. Initialize $V(s)$ arbitrarily and $N(s) \leftarrow 0$ for every state s.
2. For each episode:
   1. Generate an episode with π: $s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T$.
   2. $G \leftarrow 0$.
   3. For $t = T-1, T-2, \dots, 0$:
      1. $G \leftarrow r_t + \gamma G$.
      2. If s_t does not appear among $s_0, \dots, s_{t-1}$:
         1. $N(s_t) \leftarrow N(s_t) + 1$.
         2. $V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}\big[G - V(s_t)\big]$.

Output: $V \approx V^\pi$. Every-visit Monte Carlo prediction is the same algorithm without the test in step 2.3.2.

The inner loop runs backward through the episode, so that G always holds the return from the current time step. The test in step 2.3.2 checks whether the current visit is the first one in the episode; because we move backward, the update for a state visited several times is made only when we reach its earliest visit. Step 2.3.2.2 keeps $V(s_t)$ equal to the average of the returns seen so far, as Section 5.2 explains.

5.1.3 Why averaging works

With first-visit Monte Carlo, each episode that visits s contributes exactly one return. The episodes are independent, and what happens after the first visit to s depends only on s and not on how the agent got there; this is the Markov property. So these returns are independent samples from one distribution, the distribution of the return from s under π, with mean $V^\pi(s)$ and some variance $\sigma^2(s)$. The average $\hat{V}_n(s)$ of n such returns therefore satisfies

$$\mathbb{E}\big[\hat{V}_n(s)\big] = V^\pi(s), \qquad \mathrm{Var}\big[\hat{V}_n(s)\big] = \frac{\sigma^2(s)}{n},$$

and, by the law of large numbers, $\hat{V}_n(s) \to V^\pi(s)$ with probability 1 as $n \to \infty$ (see Appendix A for both facts). In words: the estimate is right on average, its typical error is $\sigma(s)/\sqrt{n}$, and it

eventually becomes as accurate as we like. A hundred times more episodes buy ten times more accuracy.

For the uniform policy on the Trail, the return from Start has standard deviation $\sigma(\text{Start}) \approx 0.915$ (it takes the values 1, 1.8 and 3.24 with probabilities $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{4}$), and the return from Middle has standard deviation 0.8 (it is 2 or 3.6 with equal probabilities). After 100 episodes the typical error of $\hat{V}(\text{Start})$ is $0.915/\sqrt{100} \approx 0.09$. The left panel of Figure 5.1 shows 100 independent runs of the algorithm: the spread of the estimates shrinks exactly as predicted.

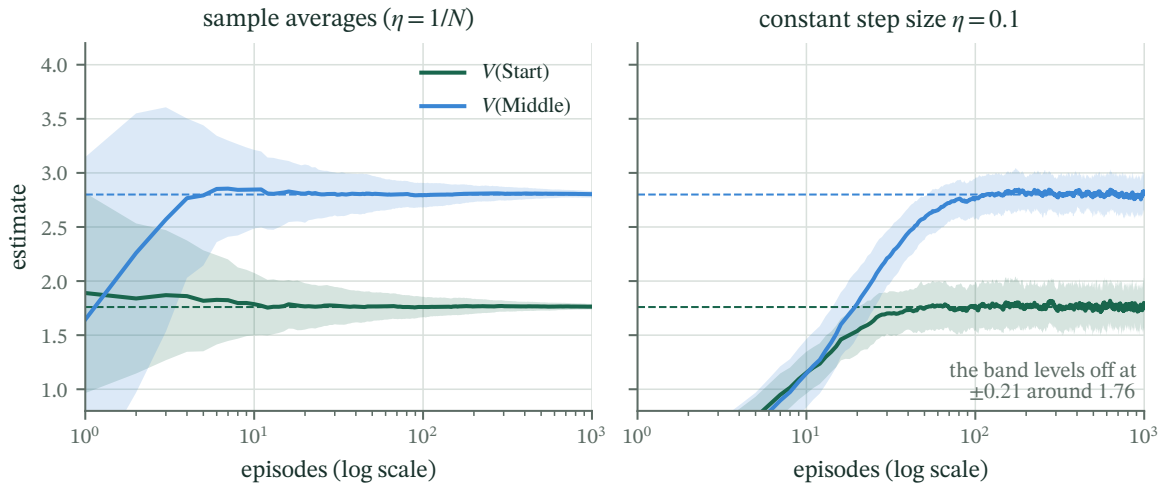


Figure 5.1. Monte Carlo prediction on the Trail (uniform policy, $\gamma = 0.9$). Lines are averages over 100 independent runs, the bands show plus or minus one standard deviation across runs, and the dashed lines mark the true values 1.76 and 2.8. Left: with sample averages the spread shrinks like $1/\sqrt{n}$. Right: with the constant step size $\eta = 0.1$ the estimates need a few dozen episodes to forget their initial value 0, and their spread does not shrink: it settles at about ±0.21.

5.1.4 What Monte Carlo methods need and what they offer

Monte Carlo prediction needs only the ability to generate episodes that end. It needs no model, and it does not *bootstrap*: the estimate for one state never uses the estimates for other states. This has two practical consequences. The cost of estimating one state's value does not depend on how many states there are, so if only a few states matter (the start state of a game, say), we can start many episodes there and ignore the rest of the state space. And errors do not propagate: a poor estimate for one state cannot corrupt the estimates of the states that lead to it.

The price is paid in waiting and in noise. Nothing can be learned until the episode ends, which rules out tasks that never end and makes long episodes slow to learn from. And the return is a sum of many random rewards, so its variance, and with it the number of episodes needed, can be large. Chapter 6 removes the waiting by bootstrapping, at the cost of some bias.

TRUNCATION IS NOT TERMINATION

Monte Carlo methods need *complete* episodes. If an episode is cut off by a time limit before it reaches a terminal state, the sum of the rewards collected so far is not a sample of the return: the rewards that would have followed are missing. Averaging such partial sums into V biases the estimates, typically toward zero. Either discard truncated episodes, or make the time limit part of the problem (by putting the remaining time into the state), or estimate the missing future with a value function, which is no longer pure Monte Carlo. Section 6.8 treats the problem in detail.

CHECKPOINT

1. In episode 4 of Table 5.1, what would the return from Middle be if $\gamma = 0.5$?
2. Why is the Monte Carlo estimate of $V(\text{Goal})$ exact after a single visit?
3. In the slippery episode above, what return does every-visit Monte Carlo use for Middle?

Answers.

1. $0 + 0.5 \times 4 = 2$.
2. Every episode that reaches Goal earns exactly 4 from there, so the return from Goal has variance 0.
3. Middle is visited once, at $t = 2$, so both methods use the same return, 2.

5.2 Incremental implementation and step sizes

Storing every return and recomputing averages is wasteful. If $\hat{V}_{n-1}$ is the average of the first $n-1$ returns $G_1, \dots, G_{n-1}$ observed from a state, the average of the first n is

$$\hat{V}_n = \frac{1}{n}\Big(G_n + (n-1)\,\hat{V}_{n-1}\Big) = \hat{V}_{n-1} + \frac{1}{n}\big(G_n - \hat{V}_{n-1}\big).$$

This is the incremental update (2.2) of Chapter 2 once more: the new estimate is the old one plus a step size times an error. The error $G_n - \hat{V}_{n-1}$ measures how much the latest return surprised the current estimate, and the step size $1/n$ says how far to move toward it. The same pattern, "old estimate plus step size times (target minus old estimate)", will carry every learning algorithm in this book; only the target changes.

Applied to the Start column of Table 5.1, with $\hat{V}_0 = 0$, the update produces the sequence 1.8, 1.4, 1.267, 1.76, 2.056, 2.013: after each episode the estimate is exactly the average of the returns so far.

Constant step sizes. A common alternative replaces $1/n$ by a constant step size $\eta \in (0, 1]$:

$$V(s_t) \leftarrow V(s_t) + \eta\,\big[G_t - V(s_t)\big]. \tag{5.1}$$

Unrolling the update shows what it computes. After n returns,

$$\hat{V}_n = (1-\eta)^n\,\hat{V}_0 + \sum_{i=1}^{n} \eta\,(1-\eta)^{n-i}\,G_i,$$

a weighted average in which the weight of a return decays geometrically with its age. Recent returns count most, the influence of the initial value fades, and returns older than a few times

$1/\eta$ are effectively forgotten. With $\eta = 0.5$ on the same six returns, the estimates are 0.9, 0.95, 0.975, 2.108, 2.674, 2.237: each new return moves the estimate halfway toward itself.

Forgetting is what we want when the quantity being estimated changes. In control (Section 5.3) the policy keeps improving, so returns observed early in learning come from a worse policy than the current one, and averaging them in forever would hold the estimates back. The price of forgetting is that the estimate never settles down. If the returns are independent with variance σ^2, the estimate of (5.1) keeps fluctuating, and its variance approaches

$$\mathrm{Var}\big[\hat{V}_n\big] \;\to\; \frac{\eta}{2-\eta}\,\sigma^2 \;\approx\; \frac{\eta}{2}\,\sigma^2 \qquad (n \to \infty).$$

For Start under the Trail's uniform policy and $\eta = 0.1$ this is $0.1 \times 0.8368/1.9 = 0.044$, a standard deviation of about 0.21: the width of the band in the right panel of Figure 5.1, which the band reaches within a few dozen episodes and then keeps instead of shrinking. Halving the step size halves the variance but doubles the time it takes to forget. Exercise 5.14 asks you to derive the formula.

The two choices are the ends of a spectrum. The step sizes $\eta_n = 1/n$ shrink fast enough to average the noise away but slowly enough to keep learning forever; Section 6.9 makes this precise with the *Robbins–Monro conditions* $\sum_n \eta_n = \infty$ and $\sum_n \eta_n^2 < \infty$. A constant step size satisfies the first condition but not the second: it keeps learning, but never stops being noisy.

5.3 Monte Carlo control

Prediction is half the problem. The other half is *control*: finding a good policy. Dynamic programming improves a policy by making it greedy with respect to its value function (Theorem 4.6), and Monte Carlo control does the same with estimated values. One adjustment is needed first.

5.3.1 Why action values

With state values, acting greedily requires a model. To choose the best action in s from V, we must compare $r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s')$ across actions, and without P and r we cannot. The remedy is to estimate *action values* $Q^\pi(s,a)$, the expected return after taking a in s and following π afterward (Definition 3.4). With Q in hand, the greedy action is simply $\arg\max_a Q(s,a)$, and no model is required.

Monte Carlo estimation of Q^π works exactly like that of V^π, with state–action pairs in place of states: a *visit* to the pair (s,a) is a time step at which the agent is in s and takes a, and the estimate of $Q^\pi(s,a)$ is the average of the returns that follow visits to (s,a).

5.3.2 Generalized policy iteration, with samples

Monte Carlo control alternates evaluation and improvement, as policy iteration does (Algorithm 4.2). It does not evaluate each policy completely, which would take infinitely many episodes. Instead it follows the pattern that Chapter 4 calls *generalized policy iteration* (Section 4.6): after every episode it updates the estimates of the visited pairs a little, and then makes

the policy greedy (or nearly greedy) with respect to the updated estimates. Evaluation chases the current policy, improvement chases the current estimates, and if all goes well both settle on the optimal ones.

5.3.3 The exploration problem

There is a catch, and the Trail shows it at once. Suppose the current policy exits at Start. Then every episode is the same: Start, exit, +1, end. The estimate of $Q(\text{Start}, \text{exit})$ becomes 1, while $Q(\text{Start}, \text{go})$ is never estimated at all, because the pair is never visited. A greedy policy keeps doing what looks best, so it never tries the actions it does not take, and it can never learn that one of them is better. Monte Carlo control must make sure that every action keeps being tried. There are two standard ways to do it.

5.3.4 Exploring starts

The first way is to change how episodes begin. Under *exploring starts*, every episode starts in a state–action pair chosen at random, with every pair having positive probability; after the first action the agent follows its current policy. Every pair is then the start of infinitely many episodes, whatever the policy does.

Algorithm 5.2 Monte Carlo control with exploring starts (MC ES)

Input: discount γ; a simulator that can start an episode in any state.

1. Initialize $Q(s,a)$ arbitrarily and $N(s,a) \leftarrow 0$ for all s, a; choose a deterministic policy π arbitrarily.
2. For each episode:
    1. Choose a start state s_0 and a first action a_0 at random, every pair with positive probability.
    2. Generate an episode from (s_0, a_0), following π afterward: $s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_{T-1}, a_{T-1}, r_{T-1}$.
    3. $G \leftarrow 0$.
    4. For $t = T-1, T-2, \dots, 0$:
        1. $G \leftarrow r_t + \gamma G$.
        2. If the pair (s_t, a_t) does not appear among $(s_0, a_0), \dots, (s_{t-1}, a_{t-1})$:
            1. $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$.
            2. $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)}\big[G - Q(s_t, a_t)\big]$.
            3. $\pi(s_t) \leftarrow \arg\max_a Q(s_t, a)$.

Output: $Q \approx Q^*$ and the greedy policy $\pi \approx \pi^*$.

Step 2.4.2.2 is first-visit Monte Carlo evaluation of Q, and step 2.4.2.3 is policy improvement in the visited states. Two practical remarks. Exploring starts require a simulator that can be put in any state, which a real robot or a real customer does not offer. And a deterministic greedy policy may cycle forever in some problems (bumping into a wall, for instance), so implementations cut episodes off after a maximum number of steps; the lab does this.

Does MC ES converge to the optimal policy? Its only possible resting point is optimal: if the policy were not optimal, the estimates of its action values would eventually reveal a better action, and the improvement step would change it. In practice it does converge. A general proof, however, is surprisingly hard to give, because the returns used to estimate $Q(s,a)$ come from a policy that keeps changing. Sutton and Barto (2018, sec. 5.3) describe the question as one of the fundamental open theoretical questions of reinforcement learning, and it has since been answered only for special cases (see Exercise 5.16).

5.3.5 On-policy control with soft policies

The second way to keep exploring needs no special starts: never let the policy become completely greedy. A policy is *ε-soft* if it gives every action at least probability $\varepsilon/|\mathcal{A}(s)|$ in every state s, where $|\mathcal{A}(s)|$ is the number of actions available in s and $\varepsilon \in (0,1]$. The *ε-greedy* policy with respect to Q, already met in Section 2.3, is the ε-soft policy that is as greedy as possible:

$$\pi(a \mid s) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a'} Q(s,a'), \\ \dfrac{\varepsilon}{|\mathcal{A}(s)|} & \text{otherwise.} \end{cases}$$

In words: with probability $1-\varepsilon$ take the greedy action, and with probability ε take an action chosen uniformly at random (which may also be the greedy one).

Algorithm 5.3 On-policy first-visit Monte Carlo control for ε-soft policies

Input: exploration rate $\varepsilon \in (0,1]$; discount γ.

1. Initialize $Q(s,a)$ arbitrarily and $N(s,a) \leftarrow 0$ for all s, a.
2. For each episode:
    1. Generate an episode with the ε-greedy policy with respect to Q: $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_{T-1}$.
    2. $G \leftarrow 0$.
    3. For $t = T-1, T-2, \dots, 0$:
        1. $G \leftarrow r_t + \gamma G$.
        2. If (s_t, a_t) does not appear among $(s_0,a_0), \dots, (s_{t-1}, a_{t-1})$:
            1. $N(s_t,a_t) \leftarrow N(s_t,a_t) + 1$.
            2. $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \frac{1}{N(s_t,a_t)}\big[G - Q(s_t,a_t)\big]$.

Output: Q and the ε-greedy policy with respect to it.

The improvement step is implicit: the next episode is generated with the ε-greedy policy with respect to the updated Q. The method is called *on-policy* because the policy it evaluates and improves is the policy that generates its data, exploratory actions included. Its target is therefore not the optimal policy but the best ε-soft policy, as the next result makes precise.

Proposition 5.1 Policy improvement for ε-soft policies

Let π be an ε-soft policy and let π' be the ε-greedy policy with respect to Q^π. Then $V^{\pi'}(s) \geq V^\pi(s)$ for every state s.

Proof. If $\varepsilon = 1$, both policies are uniform and there is nothing to prove, so let $\varepsilon < 1$. Fix s, write $m = |\mathcal{A}(s)|$, and let $a^\star$ be the greedy action. The numbers $w(a) = \big(\pi(a \mid s) - \varepsilon/m\big)/(1-\varepsilon)$ are nonnegative, because π is ε-soft, and they sum to 1. A maximum is at least any weighted average, so

$$\begin{aligned}\sum_a \pi'(a \mid s)\, Q^\pi(s,a) &= \frac{\varepsilon}{m}\sum_a Q^\pi(s,a) + (1-\varepsilon)\, Q^\pi(s,a^\star) \\ &\geq \frac{\varepsilon}{m}\sum_a Q^\pi(s,a) + (1-\varepsilon)\sum_a w(a)\, Q^\pi(s,a) = \sum_a \pi(a \mid s)\, Q^\pi(s,a) = V^\pi(s).\end{aligned}$$

So taking one step with π' and then following π is at least as good as following π throughout. Applying this inequality at every later step, exactly as in the proof of the policy improvement theorem (Theorem 4.6), gives $V^\pi(s) \leq \mathbb{E}_{\pi'}[r_0 + \gamma V^\pi(s_1) \mid s_0 = s] \leq \mathbb{E}_{\pi'}[r_0 + \gamma r_1 + \gamma^2 V^\pi(s_2) \mid s_0 = s] \leq \cdots \leq V^{\pi'}(s)$. ■

The improvement stops only when π is the best of all ε-soft policies; Sutton and Barto (2018, sec. 5.4) prove this with a neat argument that moves the forced randomness of the policy into the environment. The best ε-soft policy can be worse than the optimal policy, because it must keep taking bad actions with probability ε; on a narrow path along a cliff, an exploring agent prefers to keep its distance (Chapter 6 shows this on a real example).

5.3.6 Greedy in the limit

To learn the optimal policy rather than the best ε-soft one, exploration must fade, but slowly enough that every action is still tried infinitely often. Singh et al. (2000) call such a learning policy *greedy in the limit with infinite exploration* (GLIE):

Definition 5.2 GLIE

A sequence of learning policies is *greedy in the limit with infinite exploration* (GLIE) if (1) every state–action pair is visited infinitely often, and (2) the policy becomes greedy with respect to the current action values in the limit: the probability of a non-greedy action tends to 0.

A standard example is ε-greedy with an exploration rate that decays with the number of visits to the state, such as $\varepsilon = 1/n(s)$, where $n(s)$ counts the visits to s so far, provided that every state is visited infinitely often. Exercise 5.12 asks you to check that it is GLIE and that faster schedules such as $\varepsilon = 1/n(s)^2$ are not. GLIE is the condition under which on-policy temporal-difference control is known to converge to the optimal action values (Section 6.9). For Monte Carlo control the same schedules are used, but the convergence theory is as delicate as for MC ES.

5.3.7 An experiment on the 4 × 3 world

The *4 × 3 world* of Russell and Norvig (2021) is a small grid with a wall, a goal worth +1 and a trap worth −1 (both terminal). Every move costs 0.04, and moves are unreliable: the agent moves

in the intended direction with probability 0.8 and to each side with probability 0.1; bumping into the wall or the border leaves it in place. We use $\gamma = 0.9$. Dynamic programming gives the optimal policy shown on the right of Figure 5.2, with value $V^*(\text{start}) = 0.351$ in the bottom-left cell.

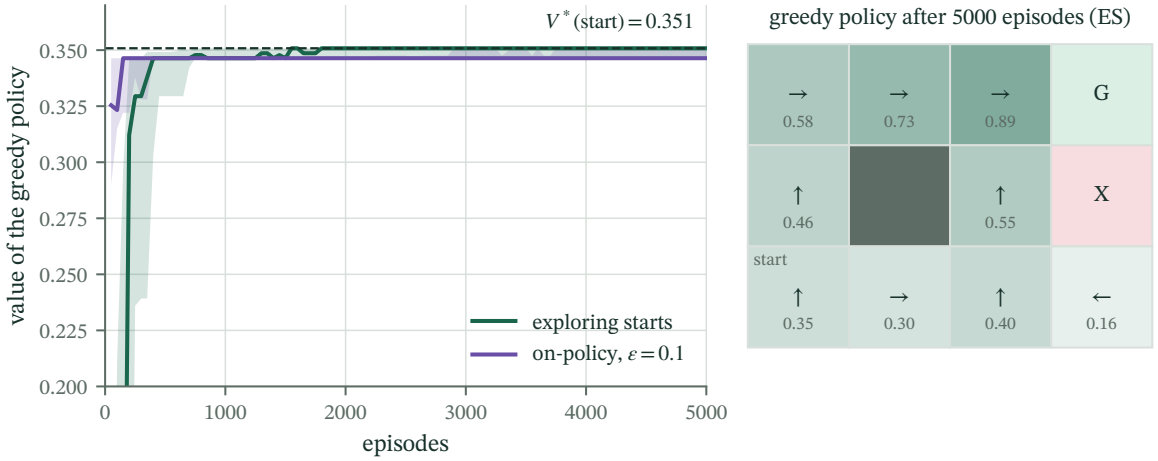


Figure 5.2. Monte Carlo control on the 4 × 3 world ($\gamma = 0.9$). Left: exact value at the start of the greedy policy after each episode, as the median over 20 runs, with bands from the 25th to the 75th percentile; exploring starts (green) and on-policy control with $\varepsilon = 0.1$ (violet). Right: the greedy policy found by exploring starts in one run after 5000 episodes, which is the optimal policy; the small numbers are the optimal values V^* computed by dynamic programming, and the shading follows them.

The left panel of Figure 5.2 follows the value of the greedy policy as learning proceeds. Both methods quickly find policies that are close to optimal. After 5000 episodes, exploring starts had found the optimal policy exactly in 13 of 20 runs, and on-policy control in 6. In most of the remaining runs the greedy policy was wrong in one to three cells and lost about 0.004 of value at the start (0.3464 instead of 0.3508); one run of each method did worse.

Why does the on-policy learner miss a few cells? The cells it gets wrong are in the bottom row, to the right of the start. Once the agent has learned to go up from the start, it reaches these cells only when a move slips, and there it tries the alternatives to its greedy action only with probability $\varepsilon \cdot \frac{3}{4}$. In one run, the action "up" in the cell next-but-one to the start was tried three times in 5000 episodes. Three returns are not enough to estimate an action value, so the greedy choice there is close to a guess. Fortunately such cells matter little: the learned policy seldom visits them, which is why its value is almost optimal. Exploring starts do not have this problem, because every pair starts episodes equally often.

KEY IDEA

Monte Carlo control is generalized policy iteration with sampled returns: estimate Q by averaging returns, act (nearly) greedily with respect to it, repeat. It works only if every action keeps being tried, which exploring starts or soft policies guarantee.

5.4 Off-policy prediction with importance sampling

The on-policy learner of the previous section has an awkward compromise built in: it learns about the policy it follows, and the policy it follows must explore. Often we would like to learn about one policy while following another. We may want to learn the greedy policy while behaving exploratorily. We may have a log of episodes recorded under an old policy, or by human operators, and want to know how a new policy would do before trying it. Learning about a *target policy* π from data generated by a different *behavior policy* b is called *off-policy learning*; learning about the policy that generates the data is *on-policy learning*.

The first requirement is obvious once stated. If the target policy can take an action that the behavior policy never takes, the data contain no information about what follows that action. We therefore assume *coverage*:

$$\pi(a \mid s) > 0 \quad \Longrightarrow \quad b(a \mid s) > 0 \qquad \text{for every state } s \text{ and action } a.$$

The target policy may be deterministic (the greedy policy, for instance), but the behavior policy must be stochastic in every state where it differs from the target.

5.4.1 The importance-sampling ratio

The returns in the data come from the wrong distribution: averaged, they estimate $V^b(s)$, not $V^\pi(s)$. *Importance sampling*, a general technique for estimating expectations under one distribution from samples drawn from another (Owen 2013, ch. 9), repairs this by reweighting each return by how much more (or less) likely its trajectory is under π than under b.

Start in s_t and follow a policy until the episode ends at time T. The probability of a particular continuation $a_t, s_{t+1}, a_{t+1}, \dots, s_T$ under π is the product of the policy's choice probabilities and the environment's transition probabilities, $\prod_{k=t}^{T-1} \pi(a_k \mid s_k)\, P(s_{k+1} \mid s_k, a_k)$, and similarly under b. Their ratio is the *importance-sampling ratio*

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(a_k \mid s_k)\, P(s_{k+1} \mid s_k, a_k)}{\prod_{k=t}^{T-1} b(a_k \mid s_k)\, P(s_{k+1} \mid s_k, a_k)} = \prod_{k=t}^{T-1} \frac{\pi(a_k \mid s_k)}{b(a_k \mid s_k)} = \prod_{k=t}^{T-1} \rho_k. \tag{5.2}$$

In words: the unknown transition probabilities cancel, because both policies act in the same environment. The ratio depends only on the two policies, which we know, and on the actions actually taken. Each factor $\rho_k = \pi(a_k \mid s_k)/b(a_k \mid s_k)$ is the per-step ratio of the notation table.

Reweighting by this ratio turns expectations under b into expectations under π. For every state s,

$$\mathbb{E}_b\big[\rho_{t:T-1}\, G_t \mid s_t = s\big] = V^\pi(s). \tag{5.3}$$

The proof is one line: the expectation under b is a sum over trajectories of b-probability times ρ times return, and b-probability times ρ is π-probability. Coverage guarantees that no trajectory with positive π-probability is missing from the sum. Exercise 5.8 asks you to write this out.

5.4.2 Ordinary and weighted importance sampling

Suppose the data contain n visits to s (first visits, one per episode, say), the i-th followed by the return G_i, with trajectory ratio ρ_i from the visit to the end of its episode. The identity (5.3) suggests averaging the reweighted returns. This is *ordinary importance sampling*:

$$\hat{V}_{\text{OIS}}(s) = \frac{1}{n}\sum_{i=1}^{n} \rho_i \, G_i. \tag{5.4}$$

A second estimator divides by the total weight instead of the number of returns. This is *weighted importance sampling*:

$$\hat{V}_{\text{WIS}}(s) = \frac{\sum_{i=1}^{n} \rho_i \, G_i}{\sum_{i=1}^{n} \rho_i}, \tag{5.5}$$

with the convention $\hat{V}_{\text{WIS}}(s) = 0$ when all the ratios are 0. The weighted estimate is a weighted average of the observed returns, with weights ρ_i; it always lies between the smallest and the largest return seen. The ordinary estimate is an average of numbers $\rho_i G_i$ that can be far larger than any return.

Example 5.1 Evaluating the optimal policy from random episodes

The six episodes of Table 5.1 were generated by the uniform policy. What do they say about the optimal policy π, which always goes on? Here b is the uniform policy, so $b(a \mid s) = \frac{1}{2}$ in Start and Middle; in Goal both policies finish, so that step's ratio is 1. There are three kinds of episodes:

Episode under b	Probability under b	Ratio $\rho_{0:T-1}$	Return G_0	$\rho_{0:T-1} G_0$
exit at Start	0.5	$0/0.5 = 0$	1	0
go, exit	0.25	$(1/0.5)(0/0.5) = 0$	1.8	0
go, go, finish	0.25	$(1/0.5)(1/0.5)(1) = 4$	3.24	12.96

Any episode containing an exit has ratio 0, because the target policy never exits. The full trip has ratio 4: it is four times as likely under π (probability 1) as under b (probability $\frac{1}{4}$).

Ordinary importance sampling is unbiased. The expected value of ρG under b is $\frac{1}{4} \times 12.96 = 3.24 = V^{\pi}(\text{Start})$, as (5.3) promises. *But it is noisy.* Each term is either 0 or 12.96, so a single term has variance $\frac{1}{4} \times 12.96^2 - 3.24^2 \approx 31.49$, a standard deviation of 5.61. For a 95% interval of half-width 0.05, the normal approximation asks for $(1.96 \times 5.61/0.05)^2 \approx 48{,}000$ episodes. On-policy, a single episode of π gives the exact answer.

Weighted importance sampling is exact here as soon as one full trip has been seen, because every episode with positive weight has the same return, 3.24.

With the six episodes of the table, two of which are full trips:

$$\hat{V}_{\text{OIS}}(\text{Start}) = \frac{12.96 + 12.96}{6} = 4.32, \qquad \hat{V}_{\text{WIS}}(\text{Start}) = \frac{12.96 + 12.96}{4 + 4} = 3.24.$$

The ordinary estimate is too high because the sample contains more full trips (two in six) than their probability ($\frac{1}{4}$) predicts.

♦

Example 5.1 is extreme in one way: the target policy is deterministic and so is the environment, so all the returns that the target policy can produce are equal. With a stochastic target the weighted estimator is not exact, but the pattern remains. Take the target policy "go on with probability 0.8 at each station", whose value is $V^\pi(\text{Start}) = 0.2\times 1 + 0.16\times 1.8 + 0.64\times 3.24 = 2.5616$, and the uniform behavior policy again. The three kinds of episodes have ratios 0.4, 0.64 and 2.56. Figure 5.3 compares the errors of the two estimators as the number of episodes grows. Weighted importance sampling has the smaller error throughout, by a factor of about three in this example.

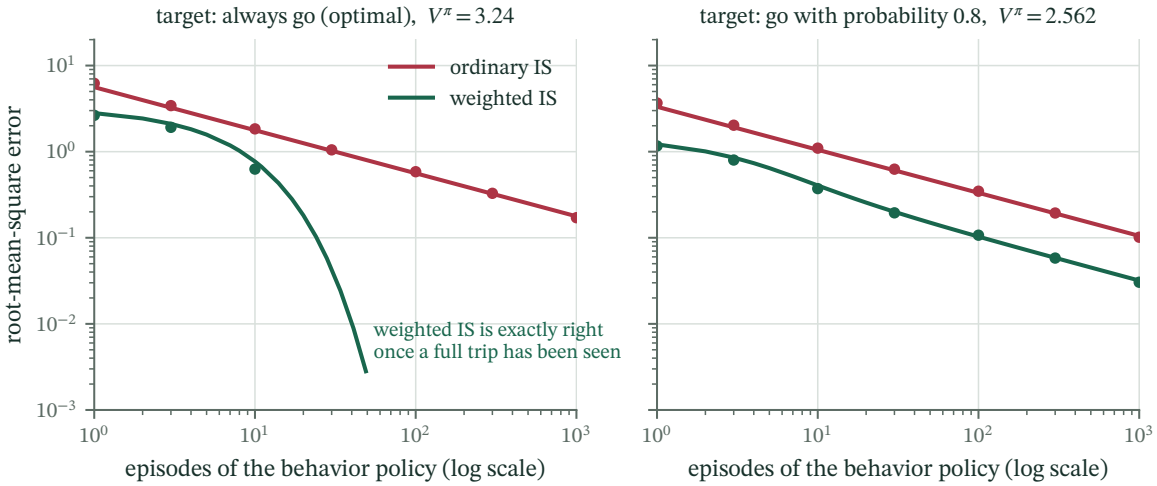


Figure 5.3. Ordinary versus weighted importance sampling on the Trail, with data from the uniform policy. Lines: exact root-mean-square errors, computed by enumerating every possible sample; dots: averages over 400 simulated runs. Left: for the optimal target, the weighted estimate is exactly right once a full trip has been seen, which happens with probability $1-(3/4)^n$ after n episodes. Right: for the target that goes on with probability 0.8, both errors fall like $1/\sqrt{n}$, and the weighted one is about three times smaller.

5.4.3 Bias and variance of the two estimators

The two estimators fail in opposite ways.

Ordinary importance sampling is unbiased: by (5.3), each term $\rho_i G_i$ has mean $V^\pi(s)$, so their average does too. Its variance, however, is governed by the ratios, which can be very large, and it can be enormous, as in Example 5.1, or even infinite (Section 5.5).

Weighted importance sampling is biased. With a single return the ratio cancels, $\hat{V}_{\text{WIS}} = \rho_1 G_1/\rho_1 = G_1$, so its expectation is $V^b(s)$, not $V^\pi(s)$: for the target that goes on with probability 0.8, the expected one-episode estimate is 1.76, the value of the uniform policy, instead of 2.5616. The bias shrinks as n grows (it is of order $1/n$), and the estimator is consistent. Its variance

is much smaller, because the estimate is an average of observed returns: if every return lies between $G_{\min}$ and $G_{\max}$, so does the estimate, whatever the ratios.

In practice the weighted estimator is usually preferred, and Mahmood et al. (2014) extend it to linear function approximation. The ordinary estimator remains important in theory, because unbiasedness makes it easy to combine and analyze, and its ideas (per-decision ratios, control variates) reappear in many later methods.

WHICH RATIO GOES WITH WHICH ESTIMATE

For a state value $V^\pi(s_t)$ the ratio is $\rho_{t:T-1}$: it starts with the ratio of the action a_t taken in s_t, because that action was drawn by the behavior policy, and it ends with the ratio of the *last* action a_{T-1}, because that action produced the last reward. For an action value $Q^\pi(s_t, a_t)$ the action a_t is given, so the ratio is $\rho_{t+1:T-1}$, without the first factor. Forgetting the last factor, or including the first one for Q, silently gives the wrong answer. On the Trail with the optimal target and uniform behavior, $Q^\pi(\text{Start}, \text{go})$ is estimated with weight 2 on the full trips: $\frac{1}{2} \times 2 \times 3.24 = 3.24$, as it should be.

5.4.4 Incremental implementation

Both estimators can be computed incrementally. Ordinary importance sampling is a plain average of the numbers $\rho_i G_i$, updated as in Section 5.2. For the weighted estimator, keep the cumulative weight $C_n = \sum_{i=1}^{n} \rho_i$; then

$$C_n = C_{n-1} + \rho_n, \qquad \hat{V}_n = \hat{V}_{n-1} + \frac{\rho_n}{C_n}\big(G_n - \hat{V}_{n-1}\big),$$

which is an incremental update with step size ρ_n / C_n in place of $1/n$. When all the ratios are 1 (on-policy data), $C_n = n$ and the two updates coincide.

5.4.5 Off-policy Monte Carlo control

Put weighted importance sampling and greedy improvement together and we obtain a control method that learns the greedy policy while behaving exploratorily. The target policy π is greedy with respect to Q, so $\pi(a \mid s)$ is 1 for the greedy action and 0 for the others. That makes the ratios simple: as we move backward through an episode, the weight is multiplied by $1/b(a_t \mid s_t)$ as long as the behavior's actions agree with the greedy ones; at the first disagreement it becomes 0, so every earlier time step gets weight 0.

Algorithm 5.4 Off-policy Monte Carlo control

Input: discount γ; a rule for choosing a soft behavior policy for each episode (for example, ε-greedy with respect to Q).

1. Initialize $Q(s, a)$ arbitrarily and $C(s, a) \leftarrow 0$ for all s, a; let $\pi(s) = \arg\max_a Q(s, a)$ in every state.
2. For each episode:

1. Choose a behavior policy b with $b(a \mid s) > 0$ for all s and a.
2. Generate an episode with b: $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_{T-1}$.
3. $G \leftarrow 0$ and $W \leftarrow 1$.
4. For $t = T-1, T-2, \dots, 0$:
   1. $G \leftarrow r_t + \gamma G$.
   2. $C(s_t, a_t) \leftarrow C(s_t, a_t) + W$.
   3. $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}\big[G - Q(s_t, a_t)\big]$.
   4. $\pi(s_t) \leftarrow \arg\max_a Q(s_t, a)$.
   5. If $a_t \neq \pi(s_t)$, stop processing this episode.
   6. $W \leftarrow W / b(a_t \mid s_t)$.

Output: $Q \approx Q^*$ and the greedy policy $\pi \approx \pi^*$.

The weight W is the ratio $\rho_{t+1:T-1}$ of the actions after a_t, as the pitfall above requires for action values. It starts at 1 for the last pair of the episode, and step 2.4.6 multiplies it by $\pi(a_t \mid s_t)/b(a_t \mid s_t) = 1/b(a_t \mid s_t)$ only after the pair (s_t, a_t) has been updated. Steps 2.4.2 and 2.4.3 are the incremental weighted average. Step 2.4.5 stops at the first action, counting from the end, that the greedy policy would not have taken.

That last step is the method's weakness: it learns only from the *tails* of episodes, the stretch after the last non-greedy action. On the Trail, with uniform behavior, it recovers Q^* exactly within a few thousand episodes. On the 4×3 world it struggles. A uniformly random behavior wanders for about 33 steps per episode, the tail that agrees with the greedy policy is usually a handful of steps near a terminal cell, and states far from the terminal cells rarely learn anything; in the lab, 3000 episodes leave the greedy policy far from optimal. Behavior policies closer to the target (an ε-greedy policy with small ε) have longer tails, and Chapter 6 removes the problem altogether.

LAB

Lab 5.1: Monte Carlo methods (`labs/ch05_mc.py`, about 45 minutes; runs in about ten seconds). Run `python3 labs/ch05_mc.py`. You will reproduce Table 5.1 and the first-visit and every-visit estimates of the slippery episode, measure the noise of sample-average and constant-step estimates against the formulas of Section 5.2, run exploring starts and on-policy control on the 4×3 world and find the rarely visited cells, compare ordinary, weighted and per-decision importance sampling, watch off-policy control learn from tails, and meet an estimator with infinite variance.

CHECKPOINT

1. A behavior policy exits at Middle with probability 1. Can it be used to evaluate the optimal policy of the Trail?
2. With target "go with probability 0.8" and uniform behavior, what is the ratio of the episode go, exit?
3. Why can a weighted importance-sampling estimate never exceed the largest observed return?

Answers.

1. No. The optimal policy goes on at Middle, an action the behavior policy never takes, so coverage fails: no episode in the data reaches Goal.
2. $\frac{0.8}{0.5} \cdot \frac{0.2}{0.5} = 1.6 \times 0.4 = 0.64$.
3. It is an average of the observed returns with nonnegative weights $\rho_i / \sum_j \rho_j$ that sum to 1.

5.5 The variance of importance sampling ●●

Example 5.1 showed that ordinary importance sampling can need tens of thousands of episodes where on-policy evaluation needs one, even in a three-state problem. This section explains why the problem grows with the length of the episodes, shows an example in which the variance is infinite, and describes estimators that reduce the variance by using the structure of the return.

5.5.1 Products of ratios

The ratio $\rho_{0:H-1}$ of an episode of H steps is a product of H per-step ratios. Each factor has mean 1 under the behavior policy, since $\mathbb{E}_b[\rho_k \mid s_k] = \sum_a b(a \mid s_k)\,\pi(a \mid s_k)/b(a \mid s_k) = 1$, but its second moment is larger than 1:

$$\mathbb{E}_b\big[\rho_k^2 \mid s_k = s\big] = \sum_a \frac{\pi(a \mid s)^2}{b(a \mid s)} = 1 + \chi^2\big(\pi(\cdot \mid s) \,\|\, b(\cdot \mid s)\big), \qquad \chi^2(p \,\|\, q) = \sum_a \frac{(p(a) - q(a))^2}{q(a)}.$$

The quantity $\chi^2(p \,\|\, q)$ is the *chi-square divergence*; it is 0 only when $p = q$. (Expand the square to check the identity.) Suppose, for simplicity, that the per-step second moment is the same number $c = 1 + \chi^2$ in every state. Conditioning on the history one step at a time (the tower rule) multiplies these factors:

$$\mathbb{E}_b\big[\rho_{0:H-1}^2\big] = c^H, \qquad \mathrm{Var}_b\big[\rho_{0:H-1}\big] = c^H - 1.$$

The variance of the ratio grows *exponentially* with the horizon, at a rate set by how different the two policies are. When c varies across states, the same argument gives the bounds $c_{\min}^H \le \mathbb{E}_b[\rho_{0:H-1}^2] \le c_{\max}^H$.

The corridor. A clean example isolates this effect. A corridor has H steps; at each step the agent picks left or right, both of which move it on and earn reward 1, and the episode ends after H steps; $\gamma = 1$. Every policy has value H, so there is nothing to learn, but an estimator does not know that. Let the target choose left with probability p and the behavior be uniform, so $c = 2\,(p^2 + (1-p)^2)$. The ordinary estimate from one episode is $H\rho_{0:H-1}$, with variance $H^2(c^H - 1)$. For the deterministic target ($p = 1$, $c = 2$) and $H = 10$ that is 102,300, and for $H = 20$ it is about 4×10^8; for a target close to the behavior ($p = 0.6$, $c = 1.04$) it is 48 at $H = 10$ and 476 at $H = 20$. Figure 5.4 shows the growth.

A useful diagnostic of the problem is Kish's *effective sample size* of a set of weights, $n_{\text{eff}} = \big(\sum_i \rho_i\big)^2 / \sum_i \rho_i^2$ (Owen 2013, ch. 9). It equals n when all weights are equal and 1 when a single weight dominates. In the corridor with a deterministic target, only episodes in which every action matched the target have positive weight, about $n/2^H$ of them, and n_{eff} is about that number.

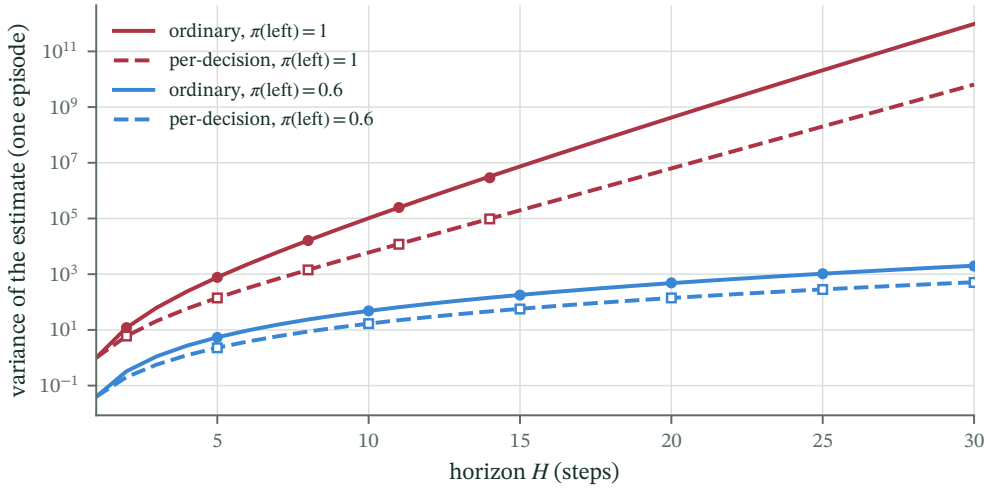


Figure 5.4. The variance of importance sampling grows exponentially with the horizon. One episode of the corridor of length H (reward 1 per step, uniform behavior), estimated with ordinary importance sampling (solid lines) and per-decision importance sampling (dashed), for a deterministic target ($\pi(\text{left}) = 1$) and a target close to the behavior ($\pi(\text{left}) = 0.6$). Lines are exact formulas; markers are sample variances from 200,000 simulated episodes. Per-decision sampling helps by a large constant factor, but the growth stays exponential.

5.5.2 When the variance is infinite

The variance can be infinite even in a problem with one state. In this example from Sutton and Barto (2018, sec. 5.5), there is a single non-terminal state s with two actions. right ends the episode with reward 0. left returns to s with probability 0.9 and otherwise ends the episode with reward +1. The target policy always goes left, so it always eventually earns 1: $V^\pi(s) = 1$ (with $\gamma = 1$). The behavior policy is uniform.

An episode has positive ratio only if all its actions were left and it ended with the +1. An episode of k such steps has probability $(0.5 \times 0.9)^{k-1}(0.5 \times 0.1)$ under b, ratio 2^k and return 1. The estimator is unbiased,

$$\mathbb{E}_b[\rho G] = \sum_{k=1}^{\infty} 0.45^{k-1} \times 0.05 \times 2^k = 0.1 \sum_{j=0}^{\infty} 0.9^j = 1,$$

but its second moment is infinite:

$$\mathbb{E}_b\big[(\rho G)^2\big] = \sum_{k=1}^{\infty} 0.45^{k-1} \times 0.05 \times 4^k = 0.2 \sum_{j=0}^{\infty} 1.8^j = \infty.$$

Long all-left episodes are rare under b, but their ratios grow faster than their probabilities shrink. Because the mean is finite, the law of large numbers still applies and the ordinary estimate converges to 1 with probability 1, but the central limit theorem does not, and the convergence is erratic. Figure 5.5 shows ten runs of a million episodes each. The weighted estimate, in contrast, is exactly 1 after the first episode with positive weight.

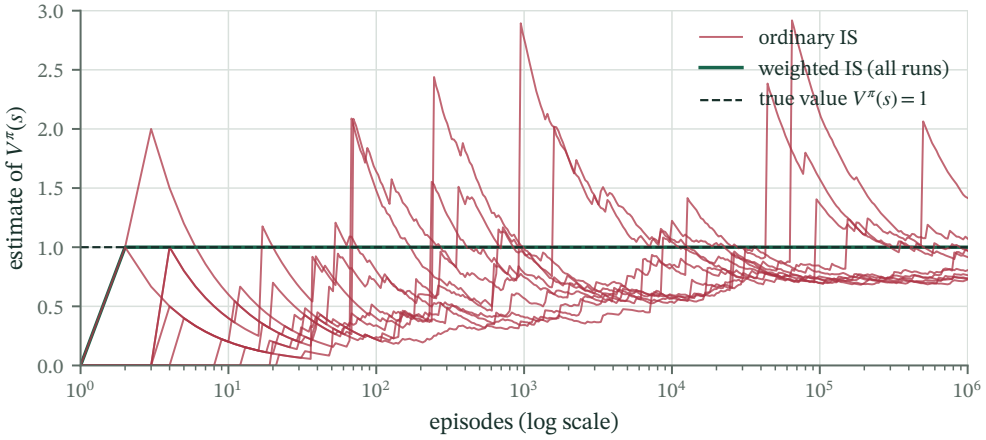


Figure 5.5. Ordinary importance sampling with infinite variance: the one-state example. Ten independent runs of the ordinary estimate (rose) over a million episodes each, on a logarithmic axis; every jump is a rare long all-left episode with a large ratio (the largest ratio in the first run is 2^{17}). The weighted estimate (green) is exactly 1 in every run as soon as one episode with positive weight has been seen.

5.5.3 Per-decision importance sampling

Ordinary importance sampling multiplies the whole return by the ratio of the whole remaining episode. Look at the return term by term,

$$\rho_{t:T-1}\, G_t = \sum_{k=t}^{T-1} \gamma^{k-t}\, \rho_{t:T-1}\, r_k,$$

and ask what each reward really needs. The reward r_k depends on the actions up to a_k, but not on the later ones, so the factors $\rho_{k+1}, \dots, \rho_{T-1}$ only add noise to the k-th term. Dropping them gives the *per-decision* return (Precup et al. 2000)

$$\tilde{G}_t = \sum_{k=t}^{T-1} \gamma^{k-t}\, \rho_{t:k}\, r_k, \qquad \hat{V}_{\mathrm{PD}}(s) = \frac{1}{n} \sum_{i=1}^{n} \tilde{G}^{(i)}, \tag{5.6}$$

where $\tilde{G}^{(i)}$ is the per-decision return of the i-th visit to s. In words: each reward is reweighted only by the ratios of the actions taken before it.

Proposition 5.3 Per-decision importance sampling is unbiased

Assume coverage, and that episodes end with probability 1 under both π and b, with integrable returns. Then $\mathbb{E}_b[\tilde{G}_t \mid s_t = s] = V^\pi(s)$ for every state s.

Proof. Fix $k \geq t$ and condition on the history up to s_{k+1}. Given that history, $\rho_{k+1:T-1}$ is the likelihood ratio of the rest of the episode, so its conditional expectation under b is the total π-probability of all continuations from s_{k+1}, which is 1 because episodes end under π (it is 1 trivially if the episode has already ended). The factor $\rho_{t:k}\, r_k$ is fixed by the history, so by the tower rule

$$\mathbb{E}_b\big[\rho_{t:T-1}\, r_k \mid s_t\big] = \mathbb{E}_b\big[\rho_{t:k}\, r_k\, \mathbb{E}_b[\rho_{k+1:T-1} \mid s_t, a_t, \dots, s_{k+1}] \mid s_t\big] = \mathbb{E}_b\big[\rho_{t:k}\, r_k \mid s_t\big].$$

Summing over k with weights γ^{k-t} gives $\mathbb{E}_b[\tilde{G}_t \mid s_t] = \mathbb{E}_b[\rho_{t:T-1} G_t \mid s_t] = V^\pi(s_t)$ by (5.3). (Exchanging the infinite sum over k and the expectation is allowed by the integrability assumption.) ■

The per-decision return has a recursive form that will matter later. Factoring out the first ratio,

$$\tilde{G}_t = \rho_t\big(r_t + \gamma\, \tilde{G}_{t+1}\big), \qquad \tilde{G}_T = 0,$$

the off-policy counterpart of $G_t = r_t + \gamma G_{t+1}$, in which each step is corrected by its own ratio. Multi-step off-policy methods (Chapter 7) start from this recursion and replace the tail $\tilde{G}_{t+1}$ by a learned estimate.

How much does it help? It depends on when the rewards arrive. In the corridor, where every step pays, the variance for the deterministic target and $H = 10$ drops from 102,300 to 5,998, but it still doubles with every extra step (Figure 5.4). On the Trail it changes nothing: every episode has a single nonzero reward, at its last step, which already needs every ratio. Per-decision weighting helps when rewards arrive along the way; it cannot help with a reward at the end.

Discounting-aware importance sampling exploits the discount in a similar way (Sutton and Barto 2018, sec. 5.8). Think of γ as a probability of continuing: the discounted return is an average of undiscounted partial returns of different lengths. With the *flat partial returns* $\bar{G}_{t:h} = r_t + r_{t+1} + \cdots + r_{h-1}$,

$$G_t = (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1}\, \bar{G}_{t:h} + \gamma^{T-t-1}\, \bar{G}_{t:T},$$

and each partial return $\bar{G}_{t:h}$ needs only the ratio $\rho_{t:h-1}$ of the actions that produced it. Weighting each term by its own ratio gives an unbiased estimator (the lab checks this by enumeration) whose variance is much smaller than that of ordinary importance sampling when the effective horizon $1/(1-\gamma)$ is short compared with the episode length.

5.5.4 Why this motivates bootstrapping

Every remedy in this section shortens the products of ratios. Per-decision sampling stops each product at the reward it corrects; discounting-aware sampling stops it at a random horizon. The extreme is to correct a single step at a time and to replace the whole remaining future by a learned estimate of its value. That is *bootstrapping*, the subject of Chapter 6. Q-learning (Chapter 6) learns about the greedy policy with no ratios at all, and multi-step methods with truncated ratios (Section 7.8) trade a little bias for bounded variance. When the data are fixed and the policies are far apart, Chapter 17 combines importance sampling with learned models ((17.1)) and with ratios of state distributions instead of trajectories (Section 17.3).

5.6 Statistical properties of Monte Carlo estimates ●●

This section proves the claims made in Section 5.1 about first-visit and every-visit estimates, compares their errors, and turns the variance of the return into confidence intervals and sample sizes.

5.6.1 First-visit estimates are unbiased

Proposition 5.4 First-visit Monte Carlo is unbiased and consistent

Let episodes be generated independently by a fixed policy π in an MDP in which episodes end with probability 1, and suppose $\mathbb{E}_\pi[|G_0| \mid s_0 = s] < \infty$. Then the returns that follow the first visits to s are independent and identically distributed, with mean $V^\pi(s)$. Consequently, given that $n \geq 1$ episodes visit s, the first-visit estimate is unbiased, has variance $\sigma^2(s)/n$ when $\sigma^2(s) = \mathrm{Var}_\pi[G_0 \mid s_0 = s]$ is finite, and converges to $V^\pi(s)$ with probability 1 as $n \to \infty$.

Proof. In one episode, let τ be the time of the first visit to s ($\tau = \infty$ if there is none). Whether $\tau = t$ is decided by $s_0, \dots, s_t$, so τ is a stopping time, and by the strong Markov property the trajectory after time τ, given $\tau < \infty$, has the same distribution as a trajectory started in s and is independent of what happened before τ. Hence the first-visit return G_τ has the distribution of G_0 given $s_0 = s$, whose mean is $V^\pi(s)$. Different episodes are independent, so the first-visit returns of the episodes that visit s are independent and identically distributed, and this remains true conditionally on which episodes visit s. The mean of n such returns has mean $V^\pi(s)$ and variance $\sigma^2(s)/n$, and the strong law of large numbers gives convergence with probability 1. ■

5.6.2 Every-visit estimates are biased but consistent

Every-visit Monte Carlo averages returns from several visits in one episode, which overlap. A one-state example shows the consequence exactly. Let a single non-terminal state s have reward 1 at every step and let the episode continue (back to s) with probability p and end with probability $1 - p$ after each step; $\gamma = 1$. The number of steps N of an episode is geometric, with $\Pr(N = k) = p^{k-1}(1 - p)$, and the true value is $V^\pi(s) = \mathbb{E}[N] = 1/(1 - p)$.

In one episode the visits are followed by the returns $N, N - 1, \dots, 1$. First-visit Monte Carlo uses N, which is unbiased. Every-visit Monte Carlo uses their average, $(N + 1)/2$, whose expectation is

$$\mathbb{E}\Big[\frac{N+1}{2}\Big] = \frac{1}{2}\Big(\frac{1}{1-p} + 1\Big) < \frac{1}{1-p} \qquad \text{for } p > 0.$$

For $p = 0.8$ (true value 5), the expected every-visit estimate from one episode is 3: a bias of -2.

With n episodes, every-visit Monte Carlo computes $\sum_i Y_i / \sum_i N_i$, where N_i is the number of visits in episode i and Y_i the sum of the returns that follow them. By the strong law of large numbers applied to the numerator and the denominator, the estimate converges to $\mathbb{E}[Y]/\mathbb{E}[N]$. Each visit's return has conditional mean $V^\pi(s)$ given that the visit occurs (by the Markov property at that time step), so $\mathbb{E}[Y] = \mathbb{E}[N]\, V^\pi(s)$, and the limit is $V^\pi(s)$. The estimator is consistent; its bias is of order $1/n$.

Which estimator has the smaller error? Figure 5.6 answers for $p = 0.8$. For up to four episodes, every-visit Monte Carlo has the smaller mean squared error: its bias costs less than its averaging of several returns gains. From five episodes on, first-visit Monte Carlo is better, and asymptotically its mean squared error is $20/n$ against about $36/n$ for every-visit (we computed both constants from the geometric distribution). Singh and Sutton (1996) proved results of this kind

for general undiscounted absorbing Markov chains: every-visit estimates are biased, first-visit estimates are unbiased, and first-visit estimates have the lower mean squared error in the long run. They derived these results while studying eligibility traces, where every-visit and first-visit Monte Carlo correspond to "accumulating" and "replacing" traces (Chapter 7).

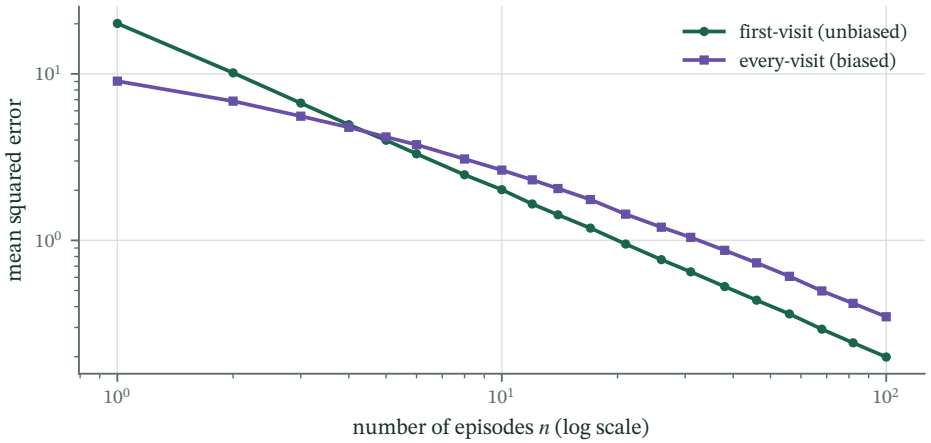


Figure 5.6. First-visit versus every-visit Monte Carlo on the one-state loop ($p = 0.8$, true value 5): mean squared error against the number of episodes, from 100,000 simulated samples of each size. Every-visit Monte Carlo, which is biased, is better for up to four episodes; first-visit Monte Carlo, which is unbiased, is better from then on.

5.6.3 Confidence intervals and sample sizes

A Monte Carlo estimate should come with an error bar. Two kinds are common.

Normal intervals. By the central limit theorem, the average of n independent returns with standard deviation σ is approximately normal, so $\hat{V} \pm 1.96\,\hat{\sigma}/\sqrt{n}$, with $\hat{\sigma}$ the sample standard deviation, contains $V^\pi(s)$ with probability about 95% when n is large. For the Trail's uniform policy and $n = 1000$ episodes, the half-width is $1.96 \times 0.915/\sqrt{1000} \approx 0.057$.

Hoeffding intervals. If every return lies in an interval of width $w = G_{\max} - G_{\min}$, Hoeffding's inequality (Theorem B.9) gives, for every n and not just approximately,

$$\Pr\Big(|\hat{V} - V^\pi(s)| \geq w\sqrt{\tfrac{\ln(2/\delta)}{2n}}\Big) \leq \delta.$$

On the Trail the returns lie in $[1, 3.24]$, so $w = 2.24$, and with $\delta = 0.05$ and $n = 1000$ the half-width is 0.096, about 1.7 times the normal one. Inverting the bound gives a sample size: $n \geq w^2 \ln(2/\delta)/(2\epsilon^2)$ episodes guarantee accuracy ϵ with probability $1 - \delta$. For $\epsilon = 0.05$ this is 3702 episodes, against 1286 from the normal approximation. Hoeffding's bound ignores the variance; Bernstein's inequality (Theorem B.11) uses it and is tighter when $\sigma \ll w$. Section B.6 collects these tools.

5.6.4 How noisy is a return?

Every sample size above is proportional to the variance of the return, which can be computed exactly when the model is known. The second moment $M^\pi(s) = \mathbb{E}_\pi[G_t^2 \mid s_t = s]$ satisfies a

Bellman equation of its own (Sobel 1982). For rewards that are deterministic functions of the state and action, expanding $(r_t + \gamma G_{t+1})^2$ gives

$$M^\pi(s) = \sum_a \pi(a \mid s)\Big[r(s,a)^2 + 2\gamma\, r(s,a) \sum_{s'} P(s' \mid s,a)\, V^\pi(s') + \gamma^2 \sum_{s'} P(s' \mid s,a)\, M^\pi(s')\Big], \quad (5.7)$$

a linear system like the Bellman equation for V^π but with discount γ^2, and $\mathrm{Var}_\pi[G_t \mid s_t = s] = M^\pi(s) - V^\pi(s)^2$. On the Trail with the uniform policy: $M(\text{Goal}) = 16$; $M(\text{Middle}) = \frac{1}{2}\times 4 + \frac{1}{2}\times 0.81 \times 16 = 8.48$, so the variance is $8.48 - 2.8^2 = 0.64$; and $M(\text{Start}) = \frac{1}{2}\times 1 + \frac{1}{2}\times 0.81 \times 8.48 = 3.9344$, so the variance is $3.9344 - 1.76^2 = 0.8368$. These are the numbers used throughout this chapter.

In general, if $|r_t| \le r_{\max}$, every return satisfies $|G_t| \le r_{\max}/(1-\gamma)$, so its standard deviation is at most of that order. The number of episodes needed for accuracy ϵ can therefore grow like $r_{\max}^2/\big((1-\gamma)^2\epsilon^2\big)$ as γ approaches 1. Long horizons are expensive for Monte Carlo methods twice over: the returns are noisier and the episodes are longer.

5.7 Monte Carlo beyond tables ●●●

Tables are the setting of this chapter, but the ideas of Monte Carlo estimation run through the whole field. This section points to three places where they reappear.

Returns as regression targets. When there are too many states for a table, we fit a parametric value function V_w instead. Monte Carlo prediction becomes regression: minimize the squared error $\sum_i \big(G_i - V_w(s_i)\big)^2$ between the observed returns and the predictions, or follow its stochastic gradient, $w \leftarrow w + \eta\big[G_t - V_w(s_t)\big]\nabla_w V_w(s_t)$. Because G_t is an unbiased sample of $V^\pi(s_t)$ that does not depend on w, this is ordinary stochastic gradient descent on the mean squared error weighted by how often states are visited, and under the usual step-size conditions it converges to a stationary point of that error (to its global minimum when V_w is linear in w). The resulting *Monte Carlo solution* is the reference point against which Chapter 9 measures the different fixed point that temporal-difference learning reaches with the same features (Section 9.4). Every-visit returns are the natural targets here.

Control variates. A classical way to reduce the variance of a Monte Carlo estimate X is to subtract a correlated quantity Y whose mean μ_Y is known: $X - c\,(Y - \mu_Y)$ has the same mean as X for every c, and its variance $\mathrm{Var}\,X - 2c\,\mathrm{Cov}(X,Y) + c^2\,\mathrm{Var}\,Y$ is smallest at $c^\star = \mathrm{Cov}(X,Y)/\mathrm{Var}\,Y$, where it equals $(1 - \mathrm{corr}(X,Y)^2)\,\mathrm{Var}\,X$ (Owen 2013, ch. 8). Importance sampling offers a natural control variate, the ratio itself, whose mean under b is 1. In Example 5.1 the optimal coefficient for $X = \rho G$ and $Y = \rho$ is $c^\star = 3.24$, and $\rho G - 3.24(\rho - 1)$ equals 3.24 for every episode: all the variance disappears. The catch is that $c^\star$ depends on unknown quantities and must be estimated, which introduces a small bias. Control variates reappear as *baselines* in policy-gradient methods (Section 11.5), where an estimate of the state's value is subtracted from the return, and as *doubly robust* off-policy estimators ((17.1)), where a learned model of Q plays the role of Y.

Monte Carlo inside modern methods. The REINFORCE policy gradient (Algorithm 11.1) weights the gradient of each action's log-probability by a Monte Carlo return. Reinforcement

learning for language models often works with a single reward at the end of a generated response, and methods such as GRPO (Algorithm 22.2) sample a group of responses to the same prompt and use the average of their rewards as the baseline, a Monte Carlo estimate of the prompt's value in place of a learned critic (Shao et al. 2024). Monte Carlo tree search (Section 8.5.3) estimates the value of a move by averaging the returns of simulated games, organized by a search tree that decides where to sample next (Kocsis and Szepesvári 2006).

RESEARCH NOTE

Open questions around Monte Carlo methods include: how to evaluate a policy off-policy over long horizons without the exponential variance of trajectory ratios (Chapter 17 surveys the main approaches); how to choose, adaptively and per state, between unbiased Monte Carlo targets and lower-variance bootstrapped ones; and when the convergence of Monte Carlo control with exploring starts can be guaranteed (Exercise 5.16).

5.8 Summary

- Monte Carlo methods estimate values by averaging the returns of complete episodes. They need no model and do not bootstrap: each state's estimate uses only the returns observed from that state.
- First-visit Monte Carlo averages independent returns, so its estimates are unbiased with error of order $\sigma/\sqrt{n}$; every-visit Monte Carlo is slightly biased but consistent, and better for very few episodes.
- Averages can be computed incrementally as "old estimate plus step size times error". A constant step size forgets old returns, which helps when the policy changes, but leaves a noise floor of variance about $\eta\sigma^2/2$.
- Control requires action values and exploration: Monte Carlo control alternates estimation of Q and greedy improvement, and it keeps exploring with exploring starts or with ε-soft policies (which converge to the best ε-soft policy; GLIE schedules let exploration fade).
- Off-policy evaluation reweights returns by the importance-sampling ratio $\rho_{t:T-1} = \prod_k \pi(a_k \mid s_k)/b(a_k \mid s_k)$, which needs no model because the transition probabilities cancel. Ordinary importance sampling is unbiased but can be extremely noisy; weighted importance sampling is biased but far more stable.
- (●●) The variance of importance sampling grows exponentially with the horizon and can be infinite; per-decision and discounting-aware sampling reduce it, but only bootstrapping removes the long products of ratios.
- (●●) Normal and Hoeffding intervals and the variance of the return, computed by its own Bellman equation, say how many episodes an estimate needs.

The next chapter keeps the model-free setting but stops waiting for the end of the episode: temporal-difference learning updates a guess from a guess after every step.

5.9 Further Reading

Sutton and Barto (2018, ch. 5) is the classic treatment of Monte Carlo methods in reinforcement learning, including blackjack examples, the infinite-variance example of Section 5.5 and discounting-aware importance sampling. For Monte Carlo estimation and importance sampling in general, Owen (2013) is an excellent, freely available book; its chapters on variance reduction and importance sampling explain control variates, self-normalized (weighted) importance sampling and effective sample sizes with care.

The statistical comparison of first-visit and every-visit Monte Carlo is due to Singh and Sutton (1996), who connect the two estimators to replacing and accumulating eligibility traces. Per-decision importance sampling and its use with eligibility traces come from Precup et al. (2000), and Mahmood et al. (2014) extend weighted importance sampling to linear function approximation. The GLIE condition is defined and used by Singh et al. (2000). The variance of the return, and its Bellman equation, go back to Sobel (1982).

On the convergence of Monte Carlo control with exploring starts, read Tsitsiklis (2002), who proves convergence of a variant in which all state–action pairs are updated at the same rate, and Wang et al. (2022), who prove convergence of the original algorithm for problems in which an optimal policy never revisits a state within an episode. The connection between Monte Carlo methods and linear algebra, which the reinforcement-learning literature exploited early, is discussed by Barto and Duff (1994).

> **HISTORICAL NOTE**
>
> The name "Monte Carlo" for computing with random samples dates from the work of Stanislaw Ulam, John von Neumann and Nicholas Metropolis at Los Alamos in the 1940s; Metropolis and Ulam (1949) is the first publication to use it. In reinforcement learning, Monte Carlo evaluation was long the baseline against which temporal-difference methods were compared, starting with Sutton (1988).

5.10 Exercises

Exercise 5.1 • Returns and estimates by hand

An agent follows some policy on the Trail with $\gamma = 0.9$ and observes four episodes: (1) go, go, finish; (2) exit; (3) go, exit; (4) go, go, finish. (a) Compute the return from every visited state in every episode. (b) Compute the first-visit Monte Carlo estimates of $V(\text{Start})$, $V(\text{Middle})$ and $V(\text{Goal})$ after each episode, with sample averages. (c) Which policy would produce these four kinds of episodes with the observed frequencies on average, and what are its true values?

Exercise 5.2 • First visit versus every visit

(a) In the one-state loop of Section 5.6 (reward 1 per step, $\gamma = 1$), you observe one episode that lasts four steps. What are the first-visit and every-visit estimates of $V(s)$? (b) On the

slippery Trail with $\gamma = 0.9$, you observe the episode Start, go (slips), Start, go (slips), Start, go, Middle, go, Goal, finish. Compute the first-visit and every-visit estimates of $V(\text{Start})$ from this episode.

Exercise 5.3 • Constant step sizes

(a) Show by induction that with a constant step size η, the estimate after n returns is $(1-\eta)^n \hat{V}_0 + \sum_{i=1}^{n} \eta(1-\eta)^{n-i} G_i$, and that the weights on $\hat{V}_0, G_1, \ldots, G_n$ sum to 1. (b) With $\eta = 0.2$ and $\hat{V}_0 = 0$, how much weight does the most recent return get, and how much do all returns older than ten episodes get together? (c) Why is $\eta = 1$ a bad choice?

Exercise 5.4 • Why Monte Carlo control must explore

An agent runs Monte Carlo control on the Trail with a greedy policy (no exploration and no exploring starts), all action values initialized to 0, and ties broken in favor of exit. (a) Describe what happens in the first three episodes and show that the agent never learns the optimal policy. (b) How would initializing every action value to 5 change the outcome? (c) Exploring starts are a third fix. Why are they easy to provide on the Trail but not for a robot learning to walk?

Exercise 5.5 • Importance-sampling ratios

The six episodes of Table 5.1 come from the uniform policy. Take the target policy that goes on with probability 0.8 in both Start and Middle. (a) Compute the ratio $\rho_{0:T-1}$ of each of the six episodes. (b) Compute the ordinary and weighted importance-sampling estimates of $V^{\pi}(\text{Start})$ and compare them with the true value 2.5616. (c) Compute both estimates of $V^{\pi}(\text{Middle})$ from the four visits to Middle (remember which ratios apply from a later time step), and compare with the true value.

Exercise 5.6 • When coverage fails

A behavior policy on the Trail goes on at Start with probability $\frac{1}{2}$ but always exits at Middle. The target policy always goes on at Start and goes on at Middle with probability $\frac{1}{2}$. (a) Show that coverage fails. (b) Find the limits of the ordinary and weighted importance-sampling estimates of $V^{\pi}(\text{Start})$ as the number of episodes grows, and compare them with the true value. (c) Explain in words why no amount of such data can reveal the true value.

Exercise 5.7 • Monte Carlo control on a grid (programming)

Using `rlcourse.mc` and the 4×3 world of Section 5.3.7: (a) run exploring starts and on-policy control ($\varepsilon = 0.1$) for 5000 episodes with several seeds and compare the greedy policies with the optimal one from `mdp.solve()`; (b) count, for one run of on-policy control, how often each action was tried in each cell, and relate the counts to the cells where the greedy action is wrong; (c) try the GLIE schedule $\varepsilon_k = 1/(1 + k/100)$ in episode k and report what changes.

Exercise 5.8 ●● Ordinary importance sampling is unbiased

(a) Write the probability of a trajectory $s_t, a_t, s_{t+1}, \dots, s_T$ under a policy, and prove the cancellation in (5.2). (b) Prove the identity (5.3) for an MDP in which every episode ends within a fixed number of steps, stating exactly where coverage is used. (c) Show that the ordinary estimate (5.4) built from first visits is unbiased.

Exercise 5.9 ●● The bias of weighted importance sampling

Use the Trail with uniform behavior and the target policy that goes on with probability 0.8 at both stations ($V^\pi(\text{Start}) = 2.5616$). (a) Show that the weighted estimate from one episode has expectation $V^b(\text{Start}) = 1.76$. (b) Enumerate the pairs of episode types and compute the exact expectation of the weighted estimate from two episodes. (c) What do your answers suggest about the size of the bias as a function of n?

Exercise 5.10 ●● When is the variance finite?

In the one-state example of Section 5.5, let left return to s with probability q instead of 0.9 (and end the episode with reward +1 otherwise); the behavior is uniform and the target always goes left. (a) Show that ordinary importance sampling is unbiased for every $q \in [0, 1)$. (b) Find the values of q for which its variance is finite, and compute the variance when it is. (c) For which q is the weighted estimate exact after the first episode with positive weight, and why?

Exercise 5.11 ●● Per-decision importance sampling

(a) Derive the recursion $\tilde{G}_t = \rho_t(r_t + \gamma \tilde{G}_{t+1})$ from the definition (5.6). (b) In the corridor of Section 5.5 with $H = 2$, deterministic target and uniform behavior, compute the variances of the ordinary and the per-decision estimates from one episode by listing the four possible episodes, and check them against the formulas behind Figure 5.4. (c) Explain why per-decision importance sampling gives exactly the same estimates as ordinary importance sampling on the Trail.

Exercise 5.12 ●● Which schedules are GLIE?

Consider ε-greedy exploration in which the exploration rate at the n-th visit to a state is ε_n, and assume every state is visited infinitely often. (a) Show that $\varepsilon_n = 1/n$ gives a GLIE learning policy. (Hint: the second Borel–Cantelli lemma.) (b) Show that constant $\varepsilon_n = 0.1$ and the schedule $\varepsilon_n = 1/n^2$ are not GLIE, and say which condition each one violates. (c) Is $\varepsilon_n = 1/\sqrt{n}$ GLIE?

Exercise 5.13 ●● The variance of the return on the slippery Trail

On the slippery Trail (each go fails with probability 0.2) with the uniform policy and $\gamma = 0.9$: (a) compute V^π for the three states by solving the Bellman equations; (b) compute the second moments M^π with (5.7) and the variance of the return from Start; (c) how many episodes does the

normal approximation require for a 95% interval of half-width 0.05 around $V^\pi(\text{Start})$? Compare with the ordinary Trail and explain the difference.

Exercise 5.14 ●● The noise floor of a constant step size

Let $G_1, G_2, \ldots$ be independent with mean μ and variance σ^2, and $\hat{V}_n = \hat{V}_{n-1} + \eta(G_n - \hat{V}_{n-1})$ with $0 < \eta < 1$. (a) Show that $\mathbb{E}[\hat{V}_n] \to \mu$ and $\mathrm{Var}[\hat{V}_n] \to \eta\sigma^2/(2-\eta)$. (b) For the Trail's uniform policy, which step size gives a stationary standard deviation of 0.05? How many episodes does it take for the influence of $\hat{V}_0$ to fall below 1%? (c) Compare with the number of episodes that sample averages need for the same accuracy.

Exercise 5.15 ●●● Importance sampling over long horizons

Off-policy evaluation over long horizons suffers from the exponential growth of Section 5.5. Survey what has been tried, organized by the idea each method uses: shortening or truncating the products of ratios; self-normalization; using a learned model as a control variate; and replacing trajectory ratios by ratios of state (or state–action) distributions. For each idea, say what it gains, what it costs (bias, assumptions), and give one reference. Which ideas need the behavior policy to be known?

Exercise 5.16 ●●● Does Monte Carlo ES converge?

(a) Explain why the only policy at which MC ES can stop changing is an optimal one, and why this does not prove convergence. (b) Read Tsitsiklis (2002) and Wang et al. (2022). State precisely what each proves, under which assumptions, and how their algorithms differ from Algorithm 5.2. (c) What property of "feed-forward" problems, in which states are not revisited within an episode, makes the argument of the second paper possible? Propose an experiment that looks for non-convergence in a problem with loops.

CHAPTER 6

Temporal-Difference Learning

WHAT YOU WILL LEARN

- Explain bootstrapping and the temporal-difference (TD) error, and perform TD(0) updates by hand.
- Compare TD and Monte Carlo prediction: bias and variance, learning during episodes, and the different answers they give on a fixed batch of data.
- Implement SARSA, Q-learning and Expected SARSA, and explain how their targets differ and which of them are on-policy or off-policy.
- Explain the cliff-walking results: why the algorithm with the better learned policy can earn less while learning.
- Explain maximization bias and how Double Q-learning removes it.
- Handle terminal states and time limits correctly when bootstrapping.
- State the convergence theorem for tabular Q-learning with its conditions, and sketch its proof by stochastic approximation (●●).
- Summarize what is known about the sample complexity of Q-learning and its dependence on $1/(1-\gamma)$ (●●●).

Monte Carlo methods wait. They let an episode run to its end, add up the rewards, and only then update their estimates. Dynamic programming does not wait, but it needs a model: it computes the value of a state from the values of the states that can follow it, weighted by their known probabilities. *Temporal-difference learning* combines the best of both. Like Monte Carlo, it learns from experience, without a model. Like dynamic programming, it updates an estimate from other estimates, without waiting for the final outcome. After every single step, it compares its prediction for the state it just left with the reward it received plus its prediction for the state it reached, and it corrects the first prediction toward the second.

This idea, *learning a guess from a guess*, is arguably the most distinctive idea of reinforcement learning. Samuel's checkers program used a form of it in 1959 (Samuel 1959); Sutton gave it its name and its first analysis (Sutton 1988); Watkins's Q-learning turned it into a method for learning optimal behavior (Watkins 1989); and it is the core of the deep Q-networks of Chapter 10 and of the critics in the actor-critic methods of Part IV. It even appears to be implemented in the brain, as Section 6.11 explains.

Learning from a guess raises obvious worries. If the guess is wrong, the learner chases a moving, biased target. When is that better than waiting for the truth, and does it converge at all? This chapter answers these questions on small problems where every number can be checked, and it ends with the theory that guarantees convergence and the research that measures how fast it is.

Section 6.1 introduces TD prediction and the TD error, and Section 6.2 compares TD with Monte Carlo, with experiments and with a precise statement of what each converges to on a fixed batch of data. Sections 6.3 to 6.5 turn TD into control methods: SARSA, Q-learning and Expected SARSA. Section 6.6 compares them on the cliff-walking task, Section 6.7 explains why Q-learning overestimates and how Double Q-learning fixes it, and Section 6.8 shows how to treat time limits, a frequent source of bugs. These sections are the Foundations material. Section 6.9 (●●) proves convergence, Section 6.10 (●●●) reviews finite-sample results, and Section 6.11 (●●) tells the story of TD and dopamine. On a first reading, read Sections 6.1 to 6.8 and the summary.

6.1 TD prediction

6.1.1 Predicting a commute

Here is an everyday version of the problem. You commute by train and try to predict how long the trip will take. You leave home at 8:00 and expect the trip to take 35 minutes. At the station, 7 minutes later, you learn that the train is 10 minutes late, and you now expect another 38 minutes, 45 in total. The train leaves at 8:17; you expect 25 more minutes, 42 in total. A signal problem holds the train outside the city, and it arrives downtown at 8:45; you expect 7 more minutes, 52 in total. A slow elevator brings the total to 56 minutes.

Table 6.1. The commute: predictions made along the way. Each row's "predicted minutes to go" is the value of that situation, and the minutes of each leg are its rewards (costs).

Situation	Minutes elapsed	Predicted minutes to go	Predicted total
leave home	0	35	35
reach station (train is late)	7	38	45
train leaves	17	25	42
arrive downtown	45	7	52
enter building	50	4	54
at desk	56	0	56

How should the predictions change in light of this trip? A Monte Carlo learner waits until you reach your desk and then moves every prediction toward what actually happened: the prediction made at home (35 minutes) toward 56, the one made at the station (45) toward 56, and so on. A TD learner moves each prediction toward the *next* prediction, as soon as that becomes available. When you reach the station and expect 45 minutes in total, it updates the home prediction toward 45, without waiting to see how the day ends. At departure it updates the station prediction toward 42, and so on down the line. Figure 6.1 shows both kinds of updates.

The TD learner has two advantages here. It can learn before the trip ends; if you never reach the office (the phone rings and the meeting is cancelled), it has still learned that late trains add time. And its updates are local: the change to the home prediction depends on what happened between

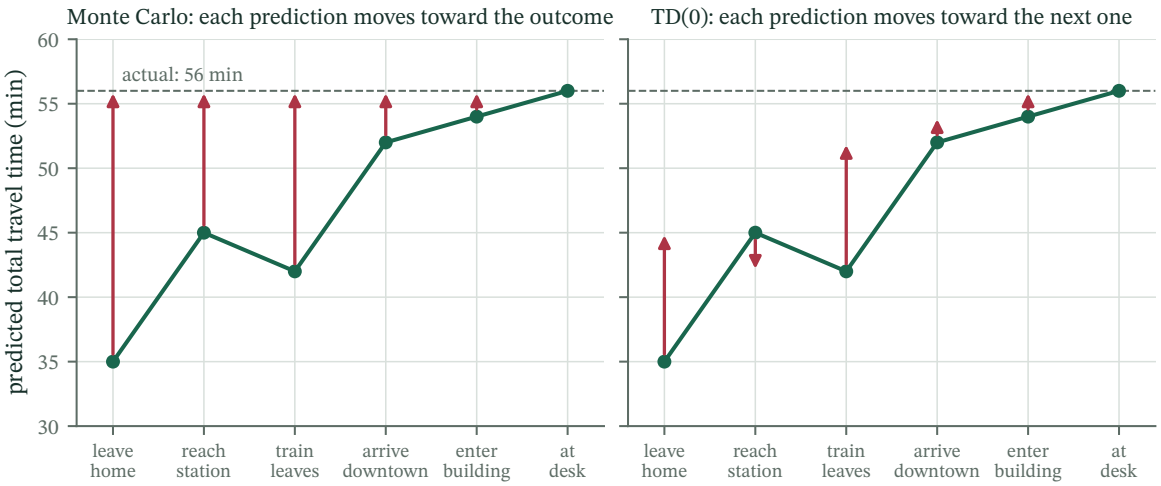


Figure 6.1. Monte Carlo and TD updates in the commute example. The line shows the predicted total travel time in each situation; the arrows show the direction in which each prediction is updated after this trip. Monte Carlo moves every prediction toward the actual total of 56 minutes. TD moves each prediction toward the one that followed it; the home prediction moves toward 45, the station prediction down toward 42.

home and the station, not on the elevator. The price is that the target, the next prediction, can itself be wrong. Whether that is a good trade is the subject of Section 6.2.

6.1.2 The TD error and TD(0)

In the notation of the book, $V(s)$ is the current estimate of the value $V^\pi(s)$ of state s under the policy π being evaluated, γ is the discount factor, and the transition at time t is (s_t, a_t, r_t, s_{t+1}). The return obeys $G_t = r_t + \gamma G_{t+1}$, and so the values obey the Bellman equation $V^\pi(s) = \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s]$ ((3.7)). Monte Carlo prediction (Chapter 5) uses the return G_t as its target. TD prediction replaces the unknown remainder G_{t+1} of the return by the current estimate $V(s_{t+1})$, and uses the *TD target* $r_t + \gamma V(s_{t+1})$. The difference between the target and the current estimate is the *temporal-difference error*, or *TD error*:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t). \tag{6.1}$$

In words: the TD error is how much better (or worse) the step turned out than the agent expected, counting both the reward that arrived and the change in the predicted future. If s_{t+1} is terminal, nothing follows it, and $V(s_{t+1})$ is taken to be 0.

The simplest TD method, *TD(0)*, moves the estimate a fraction η of the way toward the target after every step:

$$V(s_t) \leftarrow V(s_t) + \eta\, \delta_t = V(s_t) + \eta \left[r_t + \gamma V(s_{t+1}) - V(s_t)\right].$$

It is the update "old estimate plus step size times error" of (2.2) and Chapter 5 once more, with the TD target in place of the return. (The "0" in its name distinguishes it from the TD(λ) methods of Chapter 7, of which it is the special case $\lambda = 0$.)

Algorithm 6.1 Tabular TD(0) for estimating V^π

Input: the policy π to evaluate; step size $\eta \in (0, 1]$; discount γ.

1. Initialize $V(s)$ arbitrarily for all states, with $V(\text{terminal}) = 0$.
2. For each episode:
   1. Observe the initial state s.
   2. Repeat until s is terminal:
      1. Choose $a \sim \pi(\cdot \mid s)$; take it; observe r and s'.
      2. $V(s) \leftarrow V(s) + \eta\big[r + \gamma V(s') - V(s)\big]$.
      3. $s \leftarrow s'$.

Output: $V \approx V^\pi$.

TD(0) needs only the current transition: no model, no record of the episode, no wait for its end. It works unchanged in continuing tasks, which never end. In Chapter 4's terms it is a *sample* version of iterative policy evaluation (Algorithm 4.1): dynamic programming replaces $V(s)$ by the *expected* value of $r+\gamma V(s')$, computed with the model, while TD(0) moves $V(s)$ a little toward one *sample* of it.

6.1.3 Backup diagrams

The three ways of updating a value can be drawn as *backup diagrams* (Figure 3.2 introduced them for the Bellman equations). Open circles are states and dots are actions; the diagram shows which later quantities the update of the top node uses. Figure 6.2 compares them. TD(0) samples one transition and stops (it is *shallow*, and uses a *sample*). Monte Carlo samples a whole trajectory to the end (it is *deep*, and uses a sample). Dynamic programming looks one step ahead at all possible transitions (it is shallow, and uses an *expectation*). The right half of the figure anticipates the control methods of this chapter.

6.1.4 TD on the Trail, step by step

Let us run TD(0) by hand on the Trail, with $\gamma = 0.9$, step size $\eta = 0.5$ and all values starting at 0. To see the mechanics clearly we feed it four fixed episodes rather than random ones: go, go, finish; then go, exit; then the full trip again; then exit.

In the first episode the TD errors are $\delta_0 = 0 + 0.9 \times 0 - 0 = 0$ for Start, $\delta_1 = 0$ for Middle, and $\delta_2 = 4 - 0 = 4$ for Goal, whose next state is terminal. Only $V(\text{Goal})$ changes, to $0 + 0.5 \times 4 = 2$. The reward has been noticed, but its news has traveled back only one step. In the second episode Start again sees $\delta = 0 + 0.9 \times V(\text{Middle}) - 0 = 0$, and Middle, which exits with +2, sees $\delta = 2 - 0 = 2$, so $V(\text{Middle}) = 1$. In the third episode the news reaches Start: its TD error is $0 + 0.9 \times 1 - 0 = 0.9$, so $V(\text{Start}) = 0.45$; then $V(\text{Middle})$ moves toward $0.9 \times 2 = 1.8$, to 1.4, and $V(\text{Goal})$ toward 4, to 3. Table 6.2 lists the values after each episode, next to those of constant-step Monte Carlo with the same step size.

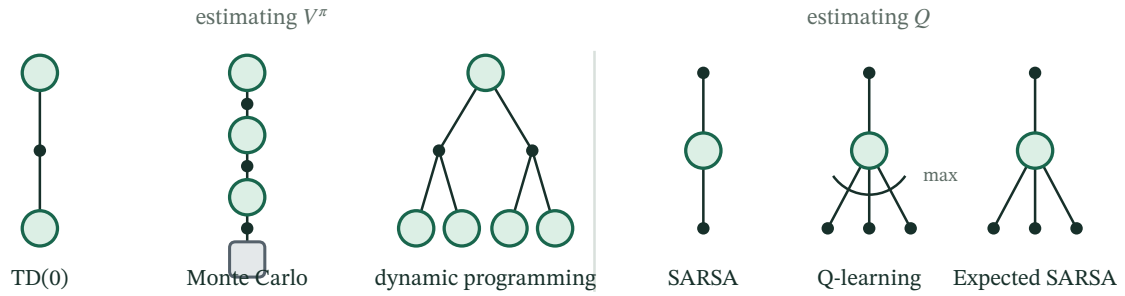


Figure 6.2. Backup diagrams. Left: three ways to update a state value: TD(0) samples one step and bootstraps from the next state's estimate; Monte Carlo samples a complete episode, to the terminal state (square); dynamic programming averages over all actions and next states with the model's probabilities. Right: the action-value methods of Sections 6.3–6.5, which start from a state–action pair (dot): SARSA bootstraps from the next action actually taken, Q-learning from the best next action (the arc marks the maximum), Expected SARSA from the average over next actions.

Table 6.2. TD(0) and constant-step Monte Carlo on four fixed episodes of the Trail ($\gamma = 0.9$, $\eta = 0.5$, initial values 0). TD propagates the reward backward one state per episode; Monte Carlo moves every visited state toward the whole return at once.

After episode	TD(0): Start	Middle	Goal	Monte Carlo: Start	Middle	Goal
1: go, go, finish	0	0	2	1.62	1.8	2
2: go, exit	0	1	2	1.71	1.9	2
3: go, go, finish	0.45	1.4	3	2.475	2.75	3
4: exit	0.725	1.4	3	1.7375	2.75	3

Monte Carlo is faster here: after one episode it has already credited Start with a share of the +4. That is typical of the first few episodes, when the estimates that TD bootstraps from are still poor. TD's estimates, on the other hand, are built from shorter, less variable pieces, and that matters more as experience accumulates.

The Monte Carlo error is a sum of TD errors. If V does not change during an episode, the TD errors add up to the Monte Carlo error:

$$G_t - V(s_t) = \sum_{k=t}^{T-1} \gamma^{k-t}\, \delta_k. \tag{6.2}$$

To see this, write $G_t - V(s_t) = \delta_t + \gamma\big(G_{t+1} - V(s_{t+1})\big)$, which holds because $G_t = r_t + \gamma G_{t+1}$, and unroll it to the end, where $G_T - V(s_T) = 0$. In the commute, the TD errors (in minutes, measured on the predicted totals) are $+10$, -3, $+10$, $+2$ and $+2$, and they add up to $+21$, the difference between the actual total of 56 minutes and the first prediction of 35. Monte Carlo makes this correction in one big step at the end; TD makes it in pieces, as the evidence arrives. The identity is the starting point of the eligibility traces of Chapter 7.

CHECKPOINT

1. In the first episode of Table 6.2, why does $V(\text{Start})$ not change?
2. What is the TD error of the transition from the station to the departure in the commute (in predicted total time)?
3. Does TD(0) need to know $P(s' \mid s, a)$?

Answers.

1. Its TD error is $0 + 0.9\,V(\text{Middle}) - V(\text{Start}) = 0$, because both estimates are still 0; the reward of this episode has not yet reached Middle's estimate.
2. $42 - 45 = -3$: the situation turned out three minutes better than predicted.
3. No. It uses only the observed transition (s, r, s') and its own estimates.

6.2 Why TD works and when it beats Monte Carlo

TD and Monte Carlo both converge to V^π under suitable step sizes (Section 6.9 proves it for TD). Which one learns faster? There is no theorem that settles the question in general, but there is a clear intuition, a classic experiment, and a precise answer for a fixed batch of data.

6.2.1 Bias and variance

The Monte Carlo target G_t is an *unbiased* sample of $V^\pi(s_t)$, but it can have a large variance, because it adds up all the randomness of the rest of the episode: every action, every transition and every reward until the end. The TD target $r_t + \gamma V(s_{t+1})$ depends on a single action, transition and reward, so its variance is usually much smaller. But it is *biased* as long as $V(s_{t+1}) \neq V^\pi(s_{t+1})$, and early in learning the estimates are far from right. TD trades variance for bias, and as the estimates improve the bias fades.

There is a second, subtler advantage. TD exploits the Markov property. When TD updates $V(s)$ toward $r + \gamma V(s')$, it uses everything it has learned about s' from every visit to s', including visits that came through other states. Monte Carlo uses, for s, only the returns that actually followed visits to s. Example 6.1 below makes this concrete.

6.2.2 The random walk

The *random walk* of Sutton and Barto (2018, Example 6.2) is a Markov reward process with five non-terminal states, A to E, in a row. Every episode starts in the middle state, C, and moves left or right with probability $\frac{1}{2}$ at each step, until it leaves the row on either side. Leaving on the

right earns +1; every other transition earns 0, and $\gamma = 1$. The value of a state is therefore the probability of leaving on the right from it, and the true values are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}$ for A to E.

Figure 6.3 compares TD(0) with constant-step Monte Carlo, both starting from the estimate 0.5 for every state. The left panel shows one run of TD(0) with $\eta = 0.1$: after 100 episodes the estimates are close to the true values. The right panel shows the root-mean-square (RMS) error over the five states, averaged over 100 runs, for several step sizes. For every step size tried, TD's error falls faster than Monte Carlo's: after 100 episodes it is 0.037 for TD with $\eta = 0.05$, against 0.082 for the best Monte Carlo curve ($\eta = 0.02$). The curves also show the cost of a constant step size. With $\eta = 0.15$, TD's error stops falling after about 30 episodes and settles at about 0.07, because each new transition keeps shaking the estimates. A smaller step size reaches a lower level, but more slowly.

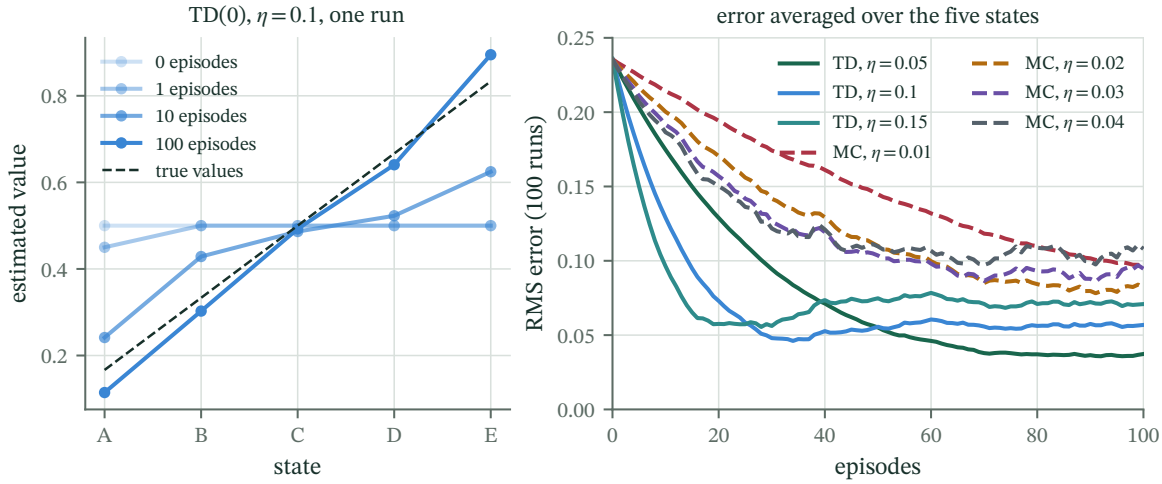


Figure 6.3. TD(0) versus constant-step Monte Carlo on the five-state random walk (initial estimates 0.5). Left: the estimates of one run of TD(0) with $\eta = 0.1$ after 0, 1, 10 and 100 episodes; after one episode only the value of the last state visited has changed. Right: root-mean-square error over the five states, averaged over 100 runs, for TD (solid) and Monte Carlo (dashed) with several step sizes. TD is better at every step size tried.

6.2.3 Batch updating

Suppose the data are limited: a fixed set of episodes, to be used as well as possible. A natural procedure is *batch updating*. Present the episodes again and again; in each pass, compute the increments that TD(0) (or Monte Carlo) would make for every transition, with V held fixed, and change V only once, by the sum of all the increments. With a small enough step size this converges, and the two methods converge to *different* answers:

- *Batch Monte Carlo* converges to the average of the returns observed from each state. These are the estimates that minimize the squared error $\sum(G - V(s))^2$ on the training data.
- *Batch TD(0)* converges to the *certainty-equivalence* estimate: the exact value function of the maximum-likelihood model of the data, the MDP whose transition probabilities and expected rewards are the observed frequencies and averages.

The second statement is easy to check. At a fixed point of batch TD, the increments from each state s add up to zero: $\sum(r + \gamma V(s') - V(s)) = 0$ over the transitions observed from s. Dividing by their number gives $V(s) = \hat{r}(s) + \gamma \sum_{s'} \hat{P}(s' \mid s)\, V(s')$, where $\hat{r}(s)$ is the average reward and $\hat{P}(s' \mid s)$ the observed frequency of s' after s. That is the Bellman equation of the estimated model. The name *certainty equivalence* refers to treating the estimated model as if it were certain. Exercise 6.14 asks for the details.

Example 6.1 You are the predictor: the A–B example

This example is due to Sutton and Barto (2018, Example 6.4). You observe eight episodes of an unknown Markov reward process with two states, A and B ($\gamma = 1$):

- once: A, reward 0, B, reward 0, end;
- six times: B, reward 1, end;
- once: B, reward 0, end.

What are the best estimates of $V(\mathrm{A})$ and $V(\mathrm{B})$? For B the answer is clear: eight episodes visited it, six of them earned 1 from there, so $V(\mathrm{B}) = \frac{6}{8} = 0.75$. Both batch methods agree.

For A they disagree. *Batch Monte Carlo* says $V(\mathrm{A}) = 0$: the only return ever observed from A was 0. *Batch TD* says $V(\mathrm{A}) = 0.75$: in the data, A always led to B with reward 0, so the maximum-likelihood model says $V(\mathrm{A}) = 0 + V(\mathrm{B}) = 0.75$.

Table 6.3. The A–B example. Batch TD coincides with certainty equivalence.

Estimate	$V(\mathrm{A})$	$V(\mathrm{B})$
batch Monte Carlo (average of returns)	0	0.75
batch TD(0)	0.75	0.75
certainty equivalence (value of the estimated model)	0.75	0.75

Which answer is better? Monte Carlo's fits the training data perfectly: it has zero squared error on the one observed return from A. But if the process is Markov, what happens after B does not depend on how we got there, and the eight observations of B say much more about the future of A than the single return that happened to follow it. TD's answer will then be more accurate on future data. If, on the contrary, the process is not Markov (reaching B through A somehow makes the reward less likely), Monte Carlo's answer is the safer one.

♦

On the random walk, batch TD is again more accurate than batch Monte Carlo: averaged over 100 sets of data, its RMS error is 0.118, 0.057 and 0.041 after 10, 50 and 100 episodes, against 0.175, 0.075 and 0.055 for batch Monte Carlo. Computing the certainty-equivalence estimate

directly, by building the model and solving its Bellman equation, takes memory of order $|\mathcal{S}|^2$ and computation of order $|\mathcal{S}|^3$; batch TD approximates it with memory of order $|\mathcal{S}|$, and online TD approaches it without storing any data at all.

> **KEY IDEA**
>
> Monte Carlo and TD answer different questions. Monte Carlo finds the estimates that best fit the observed returns; TD finds the values of the model that best explains the observed transitions. When the state is Markov, TD's answer usually generalizes better.

When to prefer Monte Carlo. TD's advantages rest on two assumptions: that the states are Markov, and that its estimates of the next states are worth trusting. When the agent's "state" leaves out relevant information (Chapter 19), bootstrapping from it propagates errors, and Monte Carlo returns, which do not depend on the state description, are more reliable. With function approximation (Chapter 9), bootstrapping can even diverge. And when the episodes are short and the rewards nearly deterministic, the variance advantage of TD is small. Chapter 7 shows how to interpolate between the two, with targets that look several steps ahead before bootstrapping.

6.3 SARSA: on-policy TD control

To turn TD prediction into a control method we follow the pattern of Chapter 5: learn action values instead of state values, so that improving the policy needs no model, and use generalized policy iteration (Section 4.6), alternating a little evaluation with a little improvement. The TD update for action values is the same as for state values, with state–action pairs in place of states. After a transition from (s_t, a_t) to s_{t+1} in which the agent then chooses a_{t+1},

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \left[r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right].$$

The update uses the five quantities $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$, which give the algorithm its name, *SARSA*. It was introduced by Rummery and Niranjan (1994), who called it modified connectionist Q-learning.

Algorithm 6.2 SARSA (on-policy TD control)

Input: step size $\eta \in (0, 1]$; exploration rate $\varepsilon > 0$; discount γ.

1. Initialize $Q(s, a)$ arbitrarily for all s, a, with $Q(\text{terminal}, \cdot) = 0$.
2. For each episode:
   1. Observe the initial state s; choose a in s ε-greedily with respect to Q.
   2. Repeat until s is terminal:
      1. Take a; observe r and s'.
      2. Choose a' in s' ε-greedily with respect to Q.
      3. $Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma Q(s', a') - Q(s, a) \right]$.
      4. $s \leftarrow s'$; $a \leftarrow a'$.

Output: Q and the ε-greedy policy with respect to it.

The next action a' is chosen in step 2.2.2 *before* the update, because the update needs it; then it is the action actually taken in the next step. Policy improvement is implicit, as in on-policy Monte Carlo control: every action is chosen ε-greedily with respect to the latest Q.

SARSA is *on-policy*: its target $r + \gamma Q(s', a')$ uses the action that the behavior policy actually takes, so it estimates the action values of that policy, exploration included. With a fixed ε it therefore learns the best ε-greedy policy, not the optimal one. To learn the optimal policy, the exploration must fade in a way that is *greedy in the limit with infinite exploration* (GLIE, defined in Chapter 5 with Algorithm 5.3): every state–action pair must still be tried infinitely often, but the probability of non-greedy actions must go to zero. Under GLIE and the step-size conditions of Section 6.9, Singh et al. (2000) proved that SARSA converges to the optimal action values with probability 1. A typical choice is ε-greedy with ε decreasing like $1/n(s)$, the inverse of the number of visits to the state.

6.4 Q-learning: off-policy TD control

Q-learning (Watkins 1989; Watkins and Dayan 1992) changes one thing in SARSA's update: instead of bootstrapping from the action actually taken next, it bootstraps from the best action available next:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right].$$

The target is a sample of the right-hand side of the Bellman optimality equation for Q^* ((3.11)), whatever policy produced the data. Q-learning therefore learns Q^* directly, while the agent behaves in any way that keeps trying every action, for example ε-greedily.

Algorithm 6.3 Q-learning (off-policy TD control)

Input: step size $\eta \in (0, 1]$; exploration rate $\varepsilon > 0$; discount γ.

1. Initialize $Q(s, a)$ arbitrarily for all s, a, with $Q(\text{terminal}, \cdot) = 0$.
2. For each episode:
    1. Observe the initial state s.
    2. Repeat until s is terminal:
        1. Choose a in s ε-greedily with respect to Q.
        2. Take a; observe r and s'.
        3. $Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$.
        4. $s \leftarrow s'$.

Output: $Q \approx Q^*$ and the greedy policy $\pi(s) = \arg\max_a Q(s, a)$.

Q-learning is *off-policy*: the policy it learns about (the greedy policy with respect to Q, the *target policy*) differs from the policy that generates the data (the ε-greedy *behavior policy*). Theorem 6.1 states the conditions under which it converges to Q^*: in essence, every state–action pair must be updated infinitely often with suitably decreasing step sizes. Nothing else is required of the behavior policy.

Why no importance sampling? Chapter 5 needed importance-sampling ratios to learn about one policy from another's data, and they caused enormous variance. One-step Q-learning needs none. The update of $Q(s, a)$ conditions on the action a, so it does not matter which policy chose it; the reward and next state come from the environment, which is the same for every policy; and the expectation over the *next* action under the target policy is not sampled at all but computed exactly, by the maximum. Nothing in the target was drawn from the behavior policy, so there is nothing to reweight. (Multi-step off-policy methods, which use several sampled actions, do need ratios or substitutes for them: Chapter 7.)

A transition on the Trail. Suppose $\gamma = 0.9$, $\eta = 0.5$, the current estimates are $Q(\text{Start}, \text{go}) = 1$, $Q(\text{Middle}, \text{exit}) = 2$ and $Q(\text{Middle}, \text{go}) = 3$, and the agent goes from Start to Middle (reward 0) and then, exploring, chooses exit. SARSA's target uses the action taken: $0 + 0.9 \times 2 = 1.8$, so $Q(\text{Start}, \text{go})$ becomes $1 + 0.5 \times 0.8 = 1.4$. Q-learning's target uses the best action: $0 + 0.9 \times 3 = 2.7$, so $Q(\text{Start}, \text{go})$ becomes $1 + 0.5 \times 1.7 = 1.85$. SARSA lowers its estimate of going on because the agent will sometimes waste the opportunity by exploring; Q-learning ignores the exploration, because the greedy policy would not waste it.

6.5 Expected SARSA and the family of one-step methods

SARSA's target contains a source of noise that can be removed: the random choice of the next action. Since the agent knows its own policy, it can replace the sampled $Q(s', a')$ by its expectation under the policy. That gives *Expected SARSA* (Seijen et al. 2009):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \Big[r_t + \gamma \sum_{a'} \pi(a' \mid s_{t+1})\, Q(s_{t+1}, a') - Q(s_t, a_t) \Big].$$

Algorithm 6.4 Expected SARSA

Input: step size $\eta \in (0, 1]$; a target policy π derived from Q (for example ε-greedy); a behavior policy (for example the same one); discount γ.

1. Initialize $Q(s, a)$ arbitrarily for all s, a, with $Q(\text{terminal}, \cdot) = 0$.
2. For each episode:
    1. Observe the initial state s.
    2. Repeat until s is terminal:
        1. Choose a in s with the behavior policy; take it; observe r and s'.
        2. $Q(s, a) \leftarrow Q(s, a) + \eta \big[r + \gamma \sum_{a'} \pi(a' \mid s')\, Q(s', a') - Q(s, a) \big]$.
        3. $s \leftarrow s'$.

Output: Q, an estimate of Q^π (of Q^* if π is greedy).

Given s_{t+1}, SARSA's target is a random sample whose expectation is Expected SARSA's target, so the two methods make the same update on average, but Expected SARSA's has lower variance. It costs a sum over the actions instead of a single lookup, which is cheap when there are few actions. Like Q-learning, Expected SARSA needs no importance sampling when its target policy differs from the behavior policy, and it is then off-policy. With an ε-greedy target policy and a single greedy action, for instance, the greedy action counts with weight $1 - \varepsilon + \varepsilon/|\mathcal{A}|$ and each other action with $\varepsilon/|\mathcal{A}|$; Example 6.2 below shows how a tie between greedy actions is handled.

The three control methods differ only in the target that the TD error compares with, as Table 6.4 summarizes.

Table 6.4. The one-step TD control methods. Expected SARSA with the behavior policy as target makes SARSA's average update with less variance; with the greedy policy as target it is Q-learning.

Method	Target for $Q(s_t, a_t)$	Learns about	Randomness in the target
SARSA	$r_t + \gamma\, Q(s_{t+1}, a_{t+1})$	the behavior policy (on-policy)	reward, next state, next action
Expected SARSA	$r_t + \gamma \sum_{a'} \pi(a' \mid s_{t+1})\, Q(s_{t+1}, a')$	any target policy π	reward, next state
Q-learning	$r_t + \gamma \max_{a'} Q(s_{t+1}, a')$	the greedy policy (off-policy)	reward, next state

The relationships are worth stating precisely. If the target policy of Expected SARSA is the behavior policy, its update is SARSA's update averaged over the next action. If the target policy is greedy with respect to Q, then $\sum_{a'} \pi(a' \mid s')\, Q(s', a') = \max_{a'} Q(s', a')$ and Expected SARSA *is* Q-learning. Expected SARSA is thus the general member of the family, and Seijen et al. (2009) show that it often performs better than SARSA, especially with large step sizes, where SARSA's extra variance hurts most.

Example 6.2 An Expected SARSA target

In some state s' with four actions the estimates are $Q(s', \cdot) = (-1, 0.5, 2, 2)$, and the policy is ε-greedy with $\varepsilon = 0.1$. The two greedy actions share the greedy probability: each has probability $\frac{1-0.1}{2} + \frac{0.1}{4} = 0.475$, and each of the other two has $\frac{0.1}{4} = 0.025$. The expected next value is $0.025 \times (-1) + 0.025 \times 0.5 + 0.475 \times 2 + 0.475 \times 2 = 1.8875$, so after a reward $r = -1$ with $\gamma = 0.9$ the Expected SARSA target is $-1 + 0.9 \times 1.8875 \approx 0.699$. SARSA's target would be $-1 + 0.9\, Q(s', a')$ for a randomly drawn a'; its variance due to the draw is $0.81 \times 0.2686 \approx 0.218$, which Expected SARSA removes.

6.6 Cliff walking

The difference between learning about the behavior policy and learning about the greedy policy shows up vividly in the *cliff-walking* task of Sutton and Barto (2018, Example 6.6).

Example 6.3 Cliff walking

The grid has 4 rows and 12 columns (Figure 6.4). The start S is in the bottom-left corner and the goal G in the bottom-right corner; the ten cells between them, along the bottom edge, are the cliff. The actions move up, down, left and right; moves off the grid leave the agent in place. Every step costs -1. Stepping into the cliff costs -100 and sends the agent back to the start, without ending the episode. Reaching the goal ends the episode. There is no discounting.

The optimal path goes up one cell, right along the edge of the cliff for eleven cells, and down into the goal: 13 steps, return -13. We ran SARSA, Q-learning and Expected SARSA with ε-greedy exploration ($\varepsilon = 0.1$), step size $\eta = 0.5$ and 500 episodes. Q-learning's greedy policy was the optimal path in all 50 of our runs. SARSA's greedy policy kept away from the edge: in 35 runs it was the *safe* path along the top row (17 steps) and in 3 runs another safe path, while in the remaining 12 runs the greedy policy read off SARSA's still-fluctuating estimates did not reach the goal at all (it walks in a loop). Expected SARSA's was the path one row above the edge (15 steps) in all runs.

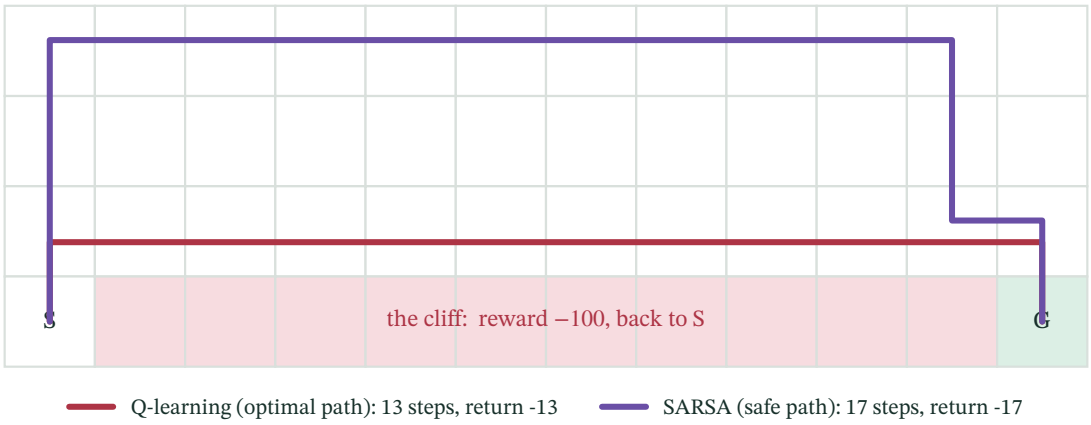


Figure 6.4. Cliff walking, with the start S at the bottom left, the goal G at the bottom right, and the cliff between them along the bottom edge. The lines are the greedy paths learned in one run (500 episodes, $\varepsilon = 0.1$, $\eta = 0.5$): Q-learning walks the edge of the cliff, the optimal path; SARSA, which learns the value of its exploring behavior, takes a safe path along the top row.

Why the difference? Q-learning learns the values of the greedy policy, which never steps off the edge, so the edge path looks best, and it is. But the agent does not *follow* the greedy policy: it explores. Walking along the edge, each step has probability $\varepsilon/4 = 0.025$ of being a random "down" into the cliff. SARSA learns the values of the policy it actually follows, exploration included, so the cells along the edge have low values for it, and it learns to keep its distance.

So which method is better? Figure 6.5 shows the sum of rewards per episode *during* learning, the *online performance*. SARSA and Expected SARSA earn much more than Q-learning: about −28 and −21 per episode over episodes 100 to 500, against −50 for Q-learning, which keeps falling off the cliff. Dynamic programming tells us where the curves are heading. The best ε-greedy policy (with $\varepsilon = 0.1$) keeps one row between itself and the cliff, and its expected return is −20.7; this is what SARSA and Expected SARSA aim at, and Expected SARSA, with less noise in its updates, gets there. The ε-greedy version of the optimal policy, which is what Q-learning executes, has expected return only −45.8. SARSA with this large step size stays somewhat below its target, because its noisier updates keep the estimates, and hence its path, fluctuating.

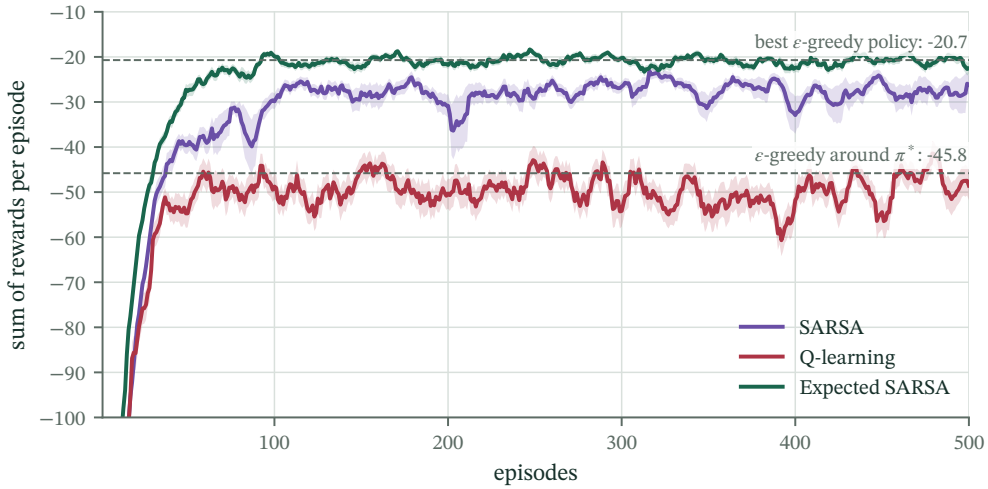


Figure 6.5. Online performance in cliff walking: sum of rewards per episode during learning ($\varepsilon = 0.1$, $\eta = 0.5$), smoothed over 10 episodes and averaged over 50 runs; bands show plus or minus one standard error. Q-learning learns the optimal path but, exploring along the edge, falls off the cliff often; SARSA and Expected SARSA learn safer paths and earn more while learning. Dashed lines: the expected returns of the best ε-greedy policy (−20.7) and of the ε-greedy version of the optimal policy (−45.8), both computed by dynamic programming.

The lesson is not that one algorithm is better, but that "better" needs a definition. If the agent will keep exploring forever (a robot that must keep learning on the job), SARSA's caution is valuable, and its on-policy values are the honest ones. If exploration will stop, and only the final greedy policy matters (training in a simulator before deployment), Q-learning's answer is the right one. If ε is decreased toward zero over time, both methods converge to the optimal policy.

PITFALL

"Q-learning learns the optimal policy" does not mean "a Q-learning agent behaves optimally". It learns the values of the greedy policy while behaving exploratorily, and its online returns can be worse than those of an on-policy learner. Always say whether you report the performance of the behavior during learning or of the greedy policy after it.

6.7 Maximization bias and Double Q-learning

Q-learning's target contains a maximum over estimated values, and that maximum is biased upward. To see why, imagine K actions whose true values are all equal to 0, and estimates $\hat{Q}(a)$ that are unbiased but noisy. The largest estimate is positive more often than not, because it picks out whichever estimate happens to have the largest positive error. In general, for any random estimates,

$$\mathbb{E}\Big[\max_a \hat{Q}(a)\Big] \;\geq\; \max_a \mathbb{E}\big[\hat{Q}(a)\big],$$

because $\max_a \hat{Q}(a) \geq \hat{Q}(a')$ for every a', and taking expectations of both sides gives $\mathbb{E}[\max_a \hat{Q}(a)] \geq \mathbb{E}[\hat{Q}(a')]$ for every a'. For independent standard normal errors the gap is substantial: the expected maximum of two such estimates is $1/\sqrt{\pi} \approx 0.56$ standard deviations, and of ten, about 1.54. In decision analysis this effect is known as the *optimizer's curse* (Smith and Winkler 2006): the alternative that looks best is, on average, worse than it looks.

In Q-learning the biased maximum enters every target, and through bootstrapping the bias propagates to the states that lead there. The result, called *maximization bias*, can mislead the agent.

A two-state example. Sutton and Barto (2018, Example 6.7) give a minimal case. Episodes start in a state A with two actions. right ends the episode with reward 0. left leads, with reward 0, to a state B, where each of several actions (ten in our experiment) ends the episode with a reward drawn from a normal distribution with mean -0.1 and standard deviation 1. Going left is worse on average (-0.1 against 0), so the optimal policy goes right. But Q-learning's estimate of going left is built from $\max_b Q(\mathrm{B}, b)$, the maximum of ten noisy estimates, which is usually positive early on. Figure 6.6 shows the result with $\varepsilon = 0.1$ and $\eta = 0.1$: Q-learning goes left in up to 94% of the episodes (around episode 20), and still in 47% after 100 episodes, although an ε-greedy agent that knew the values would go left only in the 5% of episodes in which exploration forces it.

The double estimator. The bias comes from using the same noisy estimates both to *select* the best action and to *evaluate* it. The selection favors actions whose noise is positive, and the evaluation then reports that positive noise as value. Suppose instead we have two independent sets of estimates, Q^A and Q^B. Select with one and evaluate with the other: $\hat{a} = \arg\max_a Q^A(a)$, and use $Q^B(\hat{a})$. Given Q^A, the action $\hat{a}$ is fixed and Q^B is independent of it, so $\mathbb{E}[Q^B(\hat{a}) \mid Q^A]$ is the true value of $\hat{a}$, which is at most the largest true value. The double estimator does not overestimate (it may underestimate a little; Exercise 6.13).

Double Q-learning (Hasselt 2010) applies this idea to Q-learning. It keeps two tables and, at every step, flips a coin to decide which one to update, using the other one to evaluate the action that the updated table thinks best.

Algorithm 6.5 Double Q-learning

Input: step size $\eta \in (0, 1]$; exploration rate $\varepsilon > 0$; discount γ.

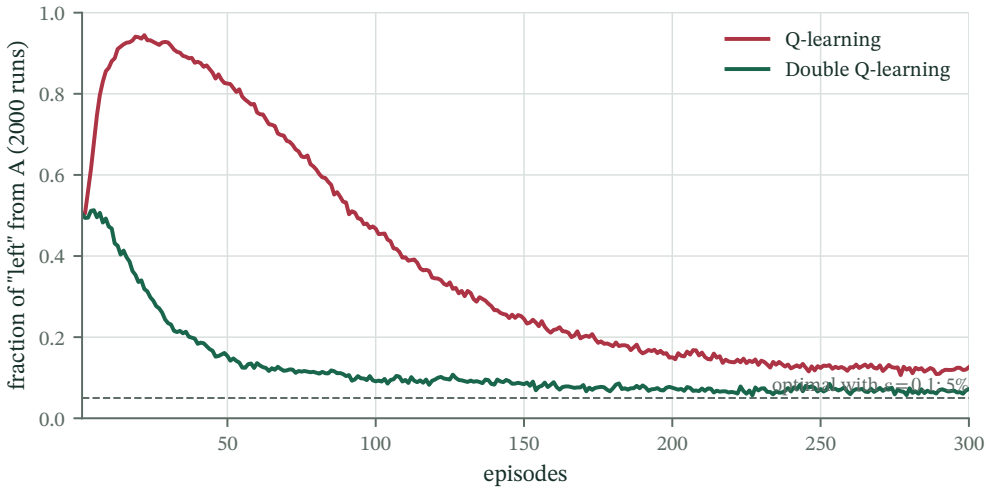


Figure 6.6. Maximization bias in the two-state example of Sutton and Barto (2018) (ten actions in B, $\varepsilon = 0.1$, $\eta = 0.1$): fraction of episodes in which the agent goes left from A, averaged over 2000 runs. Q-learning is lured to the left by the maximum of noisy estimates; Double Q-learning stays close to the 5% that ε-greedy exploration forces (dashed line).

1. Initialize $Q^A(s,a)$ and $Q^B(s,a)$ arbitrarily for all s, a, with $Q^A(\text{terminal},\cdot) = Q^B(\text{terminal},\cdot) = 0$.
2. For each episode:
   1. Observe the initial state s.
   2. Repeat until s is terminal:
      1. Choose a in s ε-greedily with respect to $Q^A + Q^B$.
      2. Take a; observe r and s'.
      3. With probability $\frac{1}{2}$: $Q^A(s,a) \leftarrow Q^A(s,a) + \eta\big[r + \gamma\, Q^B\big(s', \arg\max_{a'} Q^A(s',a')\big) - Q^A(s,a)\big]$;
      4. otherwise: $Q^B(s,a) \leftarrow Q^B(s,a) + \eta\big[r + \gamma\, Q^A\big(s', \arg\max_{a'} Q^B(s',a')\big) - Q^B(s,a)\big]$.
      5. $s \leftarrow s'$.

Output: Q^A and Q^B; act greedily with respect to their sum.

Each table is updated with half of the data, but both are used to act. In the two-state example (Figure 6.6) Double Q-learning goes left in about 9% of the episodes after 100 episodes and 7% after 300, against 47% and 13% for Q-learning. The same idea, with the online network selecting and a target network evaluating, is the Double DQN of Chapter 10 (Hasselt et al. 2016), and the related "clipped" double estimator, which takes the minimum of two critics, is a key ingredient of the TD3 algorithm of Chapter 13 (Fujimoto et al. 2018).

6.8 Terminal states, time limits and bootstrapping

Every TD target ends with a bootstrapped term, $\gamma V(s_{t+1})$, and the question of when to leave it out is the source of one of the most common bugs in reinforcement-learning code. An episode can stop for two different reasons, and they require different targets.

Termination. The episode reached a terminal state: the game is won or lost, the robot fell, the goal was reached. Nothing follows, the value of the terminal state is 0 by definition, and the target is just the reward:

$$\text{target} = r_t \qquad \text{if } s_{t+1} \text{ is terminal.}$$

Truncation. The episode was cut off for a reason outside the task: a time limit on episode length used for training, the end of a data-collection run, a reset. The state s_{t+1} is an ordinary state whose future we simply did not observe; its value is not zero. The target must keep the bootstrap:

$$\text{target} = r_t + \gamma V(s_{t+1}) \qquad \text{if the episode was truncated at } s_{t+1}.$$

The classic bug is to treat both the same way, typically through a single "done" flag that sets the bootstrap to zero whenever an episode stops. The effect is to teach the agent that the world ends at the time limit. A tiny example shows how large the error can be.

A worked example. Take a continuing task with a single state, reward +1 at every step and $\gamma = 0.9$. Its true value is $1 + 0.9 + 0.9^2 + \cdots = 1/(1 - 0.9) = 10$. Train TD(0) on episodes cut off by a time limit after five steps. With the correct targets, every transition has target $1 + 0.9\,V$, and TD converges to $V = 10$. With the bug, four of the five transitions of each episode have target $1 + 0.9\,V$ and the last one has target 1. TD converges to the point where the increments balance, $4\,(1 + 0.9V - V) + (1 - V) = 0$, that is,

$$V = \frac{5}{5 - 4 \times 0.9} \approx 3.57,$$

only about a third of the true value. The lab reproduces both numbers. With a longer limit the error is smaller but still large: with $\gamma = 0.99$ and a limit of 200 steps, the buggy fixed point is $200/(200 - 199 \times 0.99) \approx 66.9$ instead of 100.

The bug also makes the problem non-Markov from the learner's point of view: the same state sometimes has a future and sometimes, apparently, does not, depending on a clock that the agent cannot see. Pardo et al. (2018) study the issue in detail and distinguish two cases. If the time limit is only a training convenience, bootstrap at truncation, as above. If the time limit is part of the task (a game that lasts exactly 100 moves), then the remaining time matters for the best decision, and it should be part of the state; the end of time is then a genuine termination.

Modern environment interfaces make the distinction explicit. The `rlcourse` environments, like those of the Gymnasium library, return two flags from every step: `terminated` (a terminal state was reached) and `truncated` (a time limit cut the episode). The correct target uses only the first:

```
target = r if terminated else r + gamma * V[s_next]      # truncated: still bootstrap
if terminated or truncated:
    s, _ = env.reset()                                    # but start a new episode either way
```

A related trap concerns the next state itself. When an episode is truncated, the target must bootstrap from the state at which the episode was cut off, not from the first state of the next episode; vectorized environments that reset automatically must therefore keep the last observation of the old episode.

PITFALL

Treating `truncated` as `terminated` (bootstrapping from zero at a time limit) biases every value toward the values of a world that ends at the limit. Treating `terminated` as `truncated` (bootstrapping from a terminal state's estimate) is the opposite bug. Write a unit test with one transition of each kind, as the lab does.

LAB

Lab 6.1: Temporal-difference learning (`labs/ch06_td.py`, about 60 minutes; runs in about ten seconds). Run `python3 labs/ch06_td.py`. You will reproduce Table 6.2, compare TD and Monte Carlo on the random walk, solve the A–B example with batch TD, batch Monte Carlo and certainty equivalence, run SARSA, Q-learning and Expected SARSA on the cliff and compare them with the dynamic-programming references, watch maximization bias and Double Q-learning, and reproduce the time-limit bug. The "Going further" ideas include the windy grid world.

6.9 Convergence of TD and Q-learning ●●

Why should an algorithm that chases its own guesses converge, and to what? This section answers with the theory of *stochastic approximation*: TD(0) and Q-learning are noisy versions of the fixed-point iterations of Chapter 4, and noisy fixed-point iterations of contractions converge when the step sizes shrink at the right rate.

6.9.1 Stochastic approximation and the Robbins–Monro conditions

Consider the simplest case: estimating the mean μ of a random variable from samples $y_1, y_2, \ldots$, with the update $x_{n+1} = x_n + \eta_n (y_n - x_n)$. With $\eta_n = 1/n$ this computes the sample average, which converges to μ. What about other step sizes? Two forces compete. The step sizes must be large enough, in total, for the iterates to travel from any starting point to μ; and they must become small fast enough for the noise to average out. Robbins and Monro (1951) identified the conditions

$$\sum_{n=1}^{\infty} \eta_n = \infty, \qquad \sum_{n=1}^{\infty} \eta_n^2 < \infty. \tag{6.3}$$

The first condition ensures that the iterates can move arbitrarily far; the second that the accumulated noise, a sum of terms $\eta_n(y_n - \mu)$ with variance proportional to η_n^2, has finite total variance. The step sizes $\eta_n = 1/n^\omega$ satisfy both conditions exactly when $\frac{1}{2} < \omega \leq 1$ (Exercise 6.10); a constant step size satisfies only the first. The general theorem, for finding the root or fixed point of a function observed with noise, is Theorem B.1 in Appendix B.

6.9.2 TD and Q-learning as noisy fixed-point iterations

Dynamic programming iterates the Bellman operators: $V_{k+1} = \mathcal{T}^\pi V_k$ for evaluation and $Q_{k+1} = \mathcal{T}^* Q_k$ for control, where

$$(\mathcal{T}^* Q)(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a').$$

Both operators are γ-contractions in the max norm (Theorem 4.3), so the iterations converge to their unique fixed points V^π and Q^*.

Q-learning updates only one entry at a time, the visited pair (s_t, a_t), and uses one sample instead of the expectation. Write its update as

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \eta_t\Big[(\mathcal{T}^* Q_t)(s_t, a_t) - Q_t(s_t, a_t) + w_t\Big],$$
$$w_t = r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') - (\mathcal{T}^* Q_t)(s_t, a_t).$$

The noise w_t has two properties. Its conditional mean is zero: given the past and the pair (s_t, a_t), the next state is drawn from $P(\cdot \mid s_t, a_t)$ and the reward has mean $r(s_t, a_t)$, whatever the behavior policy, so $\mathbb{E}[r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') \mid \text{past}, s_t, a_t] = (\mathcal{T}^* Q_t)(s_t, a_t)$ (Exercise 6.12). And its conditional variance is bounded by $C\,(1 + \max_{s,a} Q_t(s, a)^2)$ for a constant C, if the rewards have bounded variance. So Q-learning is an *asynchronous* stochastic approximation of the fixed point of the contraction $\mathcal{T}^*$: one component at a time, observed with zero-mean noise. TD(0) is the same with $\mathcal{T}^\pi$ in place of $\mathcal{T}^*$ and states in place of state–action pairs.

6.9.3 The convergence theorem

Theorem 6.1 Convergence of tabular Q-learning

Let the MDP be finite, with discount $\gamma \in [0, 1)$ and rewards of bounded variance. At each step t let Q-learning update only the visited pair, $Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \eta_t \big[r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)\big]$, with $s_{t+1} \sim P(\cdot \mid s_t, a_t)$, and with step sizes $\eta_t \in [0, 1]$ that are determined by the history up to and including the pair (s_t, a_t). If, with probability 1, every state–action pair (s, a) satisfies

$$\sum_{t\,:\,(s_t,a_t)=(s,a)} \eta_t = \infty \qquad \text{and} \qquad \sum_{t\,:\,(s_t,a_t)=(s,a)} \eta_t^2 < \infty,$$

then $Q_t \to Q^*$ with probability 1, whatever behavior policy chooses the actions.

The first condition forces every pair to be updated infinitely often, so the behavior policy must keep trying every action in every state; the theorem asks nothing else of it. A standard choice of step size is $\eta_t = 1/n_t(s_t, a_t)^\omega$ with $\omega \in (\frac{1}{2}, 1]$, where $n_t(s, a)$ counts the updates of (s, a) so far. Convergence was first proved by Watkins and Dayan (1992); the stochastic-approximation proofs of Jaakkola et al. (1994) and Tsitsiklis (1994) showed that it is an instance of a general theorem, which also covers TD(0).

Proof. We sketch the argument; Theorem B.5 in Appendix B states the general theorem with its proof. That theorem concerns iterations $x_{t+1}(i) = x_t(i) + \eta_t(i)\big[F(x_t)(i) - x_t(i) + w_t(i)\big]$ that update one or more components i at a time ($\eta_t(i) = 0$ for the others), where F is a contraction in the max norm with fixed point x^*. It concludes that $x_t \to x^*$ with probability 1 if each component's step sizes satisfy the Robbins–Monro conditions and the noise has zero conditional mean and conditional variance at most $A + B\|x_t\|_\infty^2$. We showed above that Q-learning has exactly this form, with components (s, a), $F = \mathcal{T}^*$ (a γ-contraction by Theorem 4.3), and noise w_t with the required properties. The theorem therefore applies.

The idea behind the general theorem is worth knowing. Split the error $x_t - x^*$ into two parts that evolve separately. The noise part is a weighted sum of zero-mean noise terms with square-summable weights; a martingale argument shows that it converges to zero. Without noise, what remains is an asynchronous version of the deterministic iteration $x \leftarrow x + \eta\,(F(x) - x)$; since F shrinks distances to x^* by the factor γ, and every component keeps being updated with step sizes whose sum diverges, the max-norm error eventually falls below any multiple of its starting value. One also has to show that the iterates stay bounded, which the contraction property ensures. ■

TD(0). The same argument with $\mathcal{T}^\pi$ shows that tabular TD(0) converges to V^π with probability 1 when every state is visited infinitely often and each state's step sizes satisfy the Robbins–Monro conditions (Jaakkola et al. 1994; Tsitsiklis 1994). For episodic tasks with $\gamma = 1$, such as the random walk, the operator is a contraction in a weighted max norm provided the policy being evaluated reaches a terminal state with probability 1 from every state, and the theorem still applies (Bertsekas and Tsitsiklis 1996).

SARSA and Expected SARSA. SARSA's target depends on the next action of the behavior policy, so its expected update is $\mathcal{T}^{\pi_t}$ for the current policy π_t, which changes. If the policy is GLIE, π_t becomes greedy, $\mathcal{T}^{\pi_t}$ approaches $\mathcal{T}^*$, and Singh et al. (2000) prove that SARSA converges to Q^* with probability 1 under the same step-size conditions. With a fixed ε it cannot reach Q^*; its natural target is then the action values of the best ε-greedy policy, and Singh et al. (2000) also analyze this case of persistent exploration. Seijen et al. (2009) prove convergence of Expected SARSA under similar conditions.

Constant step sizes. With a constant step size the second Robbins–Monro condition fails, and the estimates do not converge: they keep fluctuating around the fixed point, with a spread that shrinks as the step size shrinks. On the random walk, the long-run RMS error of TD(0) with constant step size η grows roughly like $\sqrt{\eta}$ (Exercise 6.15). In practice constant step sizes are common anyway, because they track a changing target, which is what a control method must do.

6.10 Finite-sample behavior ●●●

Convergence with probability 1 says nothing about speed. How many samples does Q-learning need to be ϵ-accurate, and how does that number depend on the effective horizon $1/(1-\gamma)$? This question has been studied for three decades, and the answers depend on the setting.

Step sizes matter. Even-Dar and Mansour (2003) analyzed Q-learning with step sizes $\eta = 1/n^\omega$, where n counts the updates of a pair. With polynomial step sizes, $\omega \in (\frac{1}{2}, 1)$, the number of steps needed for ϵ-accuracy is polynomial in $1/(1-\gamma)$ (and in $1/\epsilon$ and a measure of how quickly the behavior policy covers all pairs). With the "linear" step size $1/n$, the one that computes plain averages, they show that the convergence time can be *exponential* in $1/(1-\gamma)$. The intuition: with $1/n$ steps, the early estimates, made when the bootstrapped targets were still badly wrong, are never forgotten fast enough, and the bias decays only polynomially in n at a rate that degrades as $\gamma \to 1$.

Sample complexity with a generative model. A clean setting is *synchronous* Q-learning: at each

iteration a simulator provides one sampled next state for *every* state–action pair, and all entries are updated together. Li et al. (2024) show that, with suitable step sizes, synchronous Q-learning reaches $\max_{s,a} |Q(s,a) - Q^*(s,a)| \le \epsilon$ with a total number of samples of order

$$\tilde{O}\Big(\frac{|\mathcal{S}||\mathcal{A}|}{(1-\gamma)^4\epsilon^2}\Big),$$

where $\tilde{O}$ hides logarithmic factors, and that this dependence on $1/(1-\gamma)$ is sharp *for Q-learning*: there are MDPs on which it needs that many samples. Li et al. (2024) compare this with the minimax-optimal order $|\mathcal{S}||\mathcal{A}|/\big((1-\gamma)^3\epsilon^2\big)$, up to logarithmic factors, which other algorithms attain with the same kind of data: plain Q-learning is not minimax optimal. For a single action per state, that is, for TD(0) prediction, they show that the order $(1-\gamma)^{-3}$ is attained, which is optimal.

Variance reduction. The gap can be closed. Wainwright (2019) proposed *variance-reduced Q-learning*, which, in the spirit of variance-reduced stochastic gradient methods, recenters the updates around a periodically recomputed Monte Carlo estimate of $\mathcal{T}^*$ at a reference point, and he showed that it attains the optimal $(1-\gamma)^{-3}$ dependence.

Learning from a single trajectory. In the *asynchronous* setting of Theorem 6.1, the data come from one trajectory of a behavior policy, and two more quantities enter: the smallest long-run frequency $\mu_{\min}$ with which the behavior visits a state–action pair, and the mixing time of its Markov chain. Li et al. (2020b) show a sample complexity of order $\frac{1}{\mu_{\min}(1-\gamma)^5\epsilon^2} + \frac{t_{\text{mix}}}{\mu_{\min}(1-\gamma)}$, up to logarithmic factors, and a variance-reduced variant that improves the dependence on the horizon.

RESEARCH NOTE

Open directions include: sharp finite-sample analyses of Q-learning with the constant step sizes used in practice; the behavior of Q-learning and TD with function approximation, where even convergence can fail (Section 9.8) and where overestimation takes new forms (Exercise 6.16); and the design of estimators that remove maximization bias without the underestimation of the double estimator, such as taking the minimum or the average of several estimates (Lan et al. 2020).

6.11 TD in the brain ●●

One of the most remarkable stories in the history of reinforcement learning is the discovery that the brain seems to compute something very much like the TD error.

Dopamine neurons in the midbrain of primates fire in brief bursts. In recordings from monkeys, Schultz et al. (1997) reviewed a striking pattern. When a reward, such as a drop of juice, arrives unexpectedly, the neurons burst when it arrives. When a cue, such as a light, reliably predicts the reward a second later, the burst moves, over the course of training, from the time of the reward to the time of the cue, and the reward itself no longer causes a burst. And when the predicted reward is withheld, the neurons pause their background firing at exactly the time the reward was expected.

This is the behavior of the TD error $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ of a learner that predicts reward from cues. An unexpected reward gives a positive error at the reward. Once the cue predicts the reward, the error at the reward vanishes (r_t is matched by the prediction), and the positive error moves to the appearance of the cue, whose arrival raises the prediction unexpectedly. An omitted reward gives a negative error at the time it was due. Montague et al. (1996) proposed that dopamine neurons broadcast this prediction error, and Schultz et al. (1997) presented the experimental evidence together with the model. The TD account extends the Rescorla–Wagner model of classical conditioning (Rescorla and Wagner 1972), which describes learning from one trial to the next, to learning from moment to moment within a trial; Niv (2009) reviews the evidence and its links to the models of this book.

The correspondence is not perfect, and its details are debated. Dopamine neurons also respond to novel and salient events that predict no reward, the timing of the responses depends on how the brain represents the passage of time, and different neurons appear to encode the error differently. One proposal is that they implement a distributional form of TD learning, in which different neurons learn optimistic and pessimistic predictions; Dabney et al. (2020) report evidence for this view in recordings from mice. The distributional algorithms that inspired it are the subject of Section 10.8.

6.12 Summary

- Temporal-difference learning updates an estimate toward a target that contains another estimate: $V(s_t) \leftarrow V(s_t) + \eta\,\delta_t$ with the TD error $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$. It learns from every step, without a model and without waiting for the end of the episode.
- Compared with Monte Carlo, TD's targets have lower variance but are biased while the estimates are wrong; on Markov problems such as the random walk, TD usually learns faster. On a fixed batch of data, Monte Carlo converges to the averages of the observed returns and TD to the values of the maximum-likelihood model (certainty equivalence).
- SARSA is on-policy TD control: it learns the values of the exploring policy it follows. Q-learning is off-policy: its target $r + \gamma \max_{a'} Q(s', a')$ makes it learn Q^* whatever the behavior, without importance sampling. Expected SARSA averages over the next action; it reduces SARSA's variance and contains Q-learning as the case of a greedy target policy.
- On the cliff, Q-learning learns the optimal path but earns less while exploring; SARSA learns a safe path. Which is better depends on whether performance during learning counts.
- The maximum of noisy estimates is biased upward, and this maximization bias can mislead Q-learning; Double Q-learning selects with one table and evaluates with another, which removes the overestimation.
- At a time limit, bootstrap: only true terminal states have value zero. Confusing truncation with termination is a classic and costly bug.
- (●●) Tabular TD(0) and Q-learning are asynchronous stochastic approximations of contractions; they converge with probability 1 when every pair is updated infinitely often with step sizes satisfying $\sum \eta = \infty$ and $\sum \eta^2 < \infty$. SARSA needs, in addition, a GLIE policy.
- (●●●) Synchronous Q-learning needs on the order of $|\mathcal{S}||\mathcal{A}|/((1-\gamma)^4\epsilon^2)$ samples, one factor of $1/(1-\gamma)$ more than the optimum, which variance reduction attains.

The next chapter fills the space between TD(0) and Monte Carlo with multi-step returns and eligibility traces, which let one choose how far to look ahead before bootstrapping.

6.13 Further Reading

Sutton and Barto (2018, ch. 6) is the standard introduction and the source of the random walk, the A–B example, the windy grid world, cliff walking and the maximization-bias example; its bibliographical remarks trace each idea to its origins. Sutton (1988), the paper that founded the modern theory of TD methods, is still very readable and analyzes TD under repeated presentations of a fixed training set, the batch setting of Section 6.2. Q-learning was introduced in Watkins (1989)'s thesis, and the short convergence proof of Watkins and Dayan (1992) is worth reading for its elegant "action-replay" argument. SARSA comes from Rummery and Niranjan (1994), Expected SARSA from Seijen et al. (2009), and Double Q-learning from Hasselt (2010).

For the convergence theory, Jaakkola et al. (1994) and Tsitsiklis (1994) connect TD and Q-learning with stochastic approximation; Bertsekas and Tsitsiklis (1996) give a thorough textbook treatment, and Borkar (2008) presents the ordinary-differential-equation view of stochastic approximation that underlies most modern analyses. The convergence of SARSA under GLIE exploration is due to Singh et al. (2000). Pardo et al. (2018) is the reference for the treatment of time limits.

In the ●●● direction, Even-Dar and Mansour (2003) showed how much step sizes matter, Li et al. (2024) gave the sharp sample complexity of synchronous Q-learning, Wainwright (2019) showed that variance reduction makes it minimax optimal, and Li et al. (2020b) analyzed learning from a single trajectory. On overestimation with function approximation, start with Thrun and Schwartz (1993) and continue with the double estimators of Hasselt et al. (2016) and Fujimoto et al. (2018). On TD and the brain, Schultz et al. (1997) and Montague et al. (1996) are the classic papers, Niv (2009) a careful review, and Dabney et al. (2020) a recent turn.

> **HISTORICAL NOTE**
>
> Temporal-difference ideas appear in Samuel's checkers player (Samuel 1959) and in the adaptive critic of Barto et al. (1983); Sutton (1988) named and analyzed the class of methods. Watkins's Q-learning (1989) connected them with dynamic programming, and the success of TD-Gammon (Tesauro 1995), which learned to play backgammon at the level of strong human players with TD learning and a neural network, made them famous.

6.14 Exercises

Exercise 6.1 ● TD(0) updates by hand

Run TD(0) on the Trail with $\gamma = 0.9$, step size $\eta = 0.2$ and all values initially 0, on the episodes (1) go, exit; (2) go, go, finish; (3) go, go, finish. Give every TD error and the values after each episode. Then give the values that constant-step Monte Carlo with the same step size would have after the three episodes.

Exercise 6.2 • Another batch of data

You observe four episodes of a Markov reward process with states A and B ($\gamma = 1$): once A, 0, B, 1, end; and three times B, 0, end. (a) What are the batch Monte Carlo estimates of $V(\mathrm{A})$ and $V(\mathrm{B})$? (b) What are the batch TD estimates? (c) Which would you trust if you knew that the process is Markov, and which if you suspected that coming from A changes what happens at B?

Exercise 6.3 • Three targets for one transition

In some problem, $\gamma = 0.9$ and $\eta = 0.5$. The agent is in state s with $Q(s, \cdot) = (1, 0)$, takes action 1, receives reward -1 and reaches s' with $Q(s', \cdot) = (4, 2, 1)$; its policy is ε-greedy with $\varepsilon = 0.3$, and the next action it draws is action 3. Compute the SARSA, Expected SARSA and Q-learning targets and the updated value of $Q(s, 1)$ for each method.

Exercise 6.4 • Expected SARSA's weights

(a) Show that the ε-greedy probabilities in Example 6.2 sum to 1. (b) Compute the Expected SARSA target of that example with $\varepsilon = 0$ and with $\varepsilon = 1$, and say which other algorithm each one matches. (c) Why does a tie between the greedy actions not matter for Q-learning's target but matter for the probabilities?

Exercise 6.5 • The first episode of the random walk

In the left panel of Figure 6.3, the estimates after one episode differ from the initial ones in a single state. (a) Explain why, and say by how much that estimate changed, given that $\eta = 0.1$ and all estimates started at 0.5. (b) Which state changed, and what does that tell you about how the first episode ended? (c) What would constant-step Monte Carlo with the same step size have changed after the same episode?

Exercise 6.6 • A time-limit bug

A continuing task has a single state with reward $+1$ at every step; training episodes are cut off by a time limit after H steps. (a) Show that TD(0) with the bug of Section 6.8 converges to $V = H/(H - (H - 1)\gamma)$, and that the correct targets give $1/(1 - \gamma)$. (b) Evaluate both for $\gamma = 0.9$ with $H = 5$ and $H = 50$, and for $\gamma = 0.99$ with $H = 200$. (c) Now suppose the time limit is part of the task: the episode really ends after H steps. What is the true value of the first state, what is wrong with the state description, and how would you fix it?

Exercise 6.7 • Cliff walking and exploration

(a) As ε decreases toward zero, what happens to the path that SARSA learns on the cliff, and why? (b) Explain why Q-learning's online return is worse than SARSA's although its greedy path is shorter. (c) Suppose a robot must learn on the job, falling off the "cliff" costs a repair, and exploration will never be switched off. Which method would you use, and why?

Exercise 6.8 ● The windy grid world (programming)

The windy grid world of Sutton and Barto (2018, Example 6.5) is implemented as `rl-course.td.WindyGridWorld`: a 7×10 grid, start at row 3, column 0, goal at row 3, column 7, reward -1 per step, and an upward wind of strength $0, 0, 0, 1, 1, 1, 2, 2, 1, 0$ in the ten columns. (a) Find the length of the shortest path by breadth-first search. (b) Run SARSA with $\eta = 0.5$, $\varepsilon = 0.1$ and $Q = 0$ initially, and plot the number of completed episodes against the number of time steps. How many time steps do the first 170 episodes take? (c) Repeat with king's moves (`king=True`) and with a stochastic wind (`stochastic=True`), and compare.

Exercise 6.9 ●● The maximum of noisy estimates

(a) Let X_1, X_2 be independent normal with mean 0 and variance 1. Show that $\mathbb{E}[\max(X_1, X_2)] = 1/\sqrt{\pi}$. (Hint: $\max(x, y) = \frac{1}{2}(x + y) + \frac{1}{2}|x - y|$.) (b) Now let the means be 0 and -0.5 (variances still 1). Compute $\mathbb{E}[\max(X_1, X_2)]$, using the fact that for $X_1 - X_2$ normal with mean d and standard deviation θ, $\mathbb{E}[\max(X_1, X_2)] = \mu_1 \Phi(d/\theta) + \mu_2 \Phi(-d/\theta) + \theta\, \varphi(d/\theta)$, where Φ and φ are the standard normal distribution function and density. (c) For the same two means, compute the expectation of the double estimator, which selects with one independent sample and evaluates with another, and compare its bias with that of (b).

Exercise 6.10 ●● Robbins–Monro step sizes

(a) For which $\omega > 0$ do the step sizes $\eta_n = 1/n^\omega$ satisfy both conditions (6.3)? (b) Show that for $\eta_n = 1/n$ the iteration $x_{n+1} = x_n + \eta_n(y_n - x_n)$ with x_1 arbitrary gives $x_{n+1} = \frac{1}{n}\sum_{i \le n} y_i$, and explain why the initial value disappears. (c) In Q-learning the step size of a pair should depend on the number of updates of that pair, not on the global time t. Give an example in which using $\eta_t = 1/t$ for all pairs violates the first condition for some pair.

Exercise 6.11 ●● What TD(0) converges to

(a) For a fixed policy π in a finite MDP, show that the expected TD(0) update of every state is zero if and only if V satisfies the Bellman equation $V = \mathcal{T}^\pi V$, so that the only possible limit of TD(0) is V^π. (b) Prove the identity (6.2), stating where you use the assumption that V does not change during the episode. (c) Verify it on the commute of Table 6.1.

Exercise 6.12 ●● Q-learning's expected update

(a) Show that $\mathbb{E}\big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') \mid s_t = s, a_t = a\big] = (\mathcal{T}^* Q)(s, a)$ for any fixed Q, and explain why the behavior policy does not appear. (b) Prove that $\mathcal{T}^*$ is a γ-contraction in the max norm, using $|\max_a f(a) - \max_a g(a)| \le \max_a |f(a) - g(a)|$. (c) Does the same argument work for SARSA's target? What is its conditional expectation?

Exercise 6.13 ●● The double estimator does not overestimate

Let $\mu_1, \dots, \mu_K$ be true values, and Q^A, Q^B two independent vectors of unbiased estimates of them. (a) Prove that $\mathbb{E}[Q^B(\hat{a})] \le \max_a \mu_a$ for $\hat{a} = \arg\max_a Q^A(a)$. (b) Give an example in which the inequality is strict, and compute the bias. (c) Explain why Double Q-learning can still overestimate a little in an MDP, even though the double estimator does not (hint: the two tables are not exactly independent).

Exercise 6.14 ●● Batch TD is certainty equivalence

(a) Show that the fixed points of batch TD(0) on a set of episodes are exactly the solutions of the Bellman equations of the maximum-likelihood model, and that this solution is unique when $\gamma < 1$ or when every observed state has a path to termination in the data. (b) Show that batch constant-step Monte Carlo converges to the average of the observed returns from each state. (c) For which step sizes does the batch TD iteration converge? Answer for the A–B data of Example 6.1.

Exercise 6.15 ●● The noise floor of constant step sizes (programming)

Run TD(0) on the random walk with constant step sizes $\eta \in \{0.02, 0.05, 0.1, 0.2\}$ for 1000 episodes and measure the long-run RMS error (average the error over episodes 500 to 1000, and over 20 runs). (a) How does it scale with η? (b) Give a heuristic argument for the scaling, based on Chapter 5's formula for the stationary variance of a constant-step average. (c) Propose a step-size schedule that gets the best of both worlds, and test it.

Exercise 6.16 ●●● Overestimation with function approximation

Why does Q-learning overestimate when Q is represented by a function approximator, even if the environment is deterministic? Read Thrun and Schwartz (1993) and explain their argument: approximation errors act like zero-mean noise, and the maximum in the target turns them into a systematic upward bias that bootstrapping propagates. How does the overestimation scale with the number of actions and with γ? Then discuss the remedies of Hasselt et al. (2016), Fujimoto et al. (2018) and Lan et al. (2020), and what each costs.

Exercise 6.17 ●●● Where does the extra horizon factor come from?

Li et al. (2024) show that synchronous Q-learning needs on the order of $|\mathcal{S}||\mathcal{A}|/((1-\gamma)^4\epsilon^2)$ samples, while TD learning, and variance-reduced Q-learning (Wainwright 2019), need only $(1-\gamma)^{-3}$. Explain, at the level of a careful sketch, where the extra factor comes from: how do the variance of the targets, the step sizes and the propagation of errors through γ combine? Why does variance reduction help? What changes when the data come from a single trajectory (Li et al. 2020b)?

Exercise 6.18 ●●● What does dopamine compute?

Read Schultz et al. (1997) and Niv (2009). (a) For a simple experiment in which a light predicts juice one second later, sketch the TD error over time before learning, after learning, and on a trial in which the juice is omitted, and compare with the recordings described in Section 6.11. (b) Name two observations that the basic TD account does not explain, and explain what extension has been proposed for one of them. (c) Read Dabney et al. (2020) and describe what experiment would distinguish a distributional code from a classical one.

CHAPTER 7

Multi-Step Methods and Eligibility Traces

WHAT YOU WILL LEARN

- Explain how n-step returns interpolate between one-step temporal-difference learning and Monte Carlo, and why an intermediate n often learns fastest.
- Compute n-step returns, λ-returns and eligibility traces by hand for short episodes.
- Implement n-step TD, n-step SARSA and TD(λ) with accumulating, replacing and dutch traces.
- Correct multi-step targets for off-policy data, with importance sampling or with tree backup, and explain what each costs.
- Prove that the forward and backward views of TD(λ) agree offline, and explain what true online TD(λ) adds (●●).
- Extend eligibility traces to control with SARSA(λ), Watkins's Q(λ) and Peng's Q(λ) (●●).
- Analyze off-policy traces such as Retrace and V-trace in terms of contraction, variance and fixed points (●●●).

Monte Carlo methods (Chapter 5) wait until an episode is over and then move each visited state toward the return that actually followed it. Temporal-difference methods (Chapter 6) do not wait: after a single step they move the estimate toward the reward plus the estimated value of the next state. Both are special cases of one idea, and neither extreme is best in general. Monte Carlo targets are unbiased but noisy, because one return adds up the randomness of a whole episode. One-step TD targets are steadier, but they lean on estimates that may be wrong, and they pass information backward only one step at a time.

The Trail (Section 1.3) shows the second weakness plainly. Suppose a hiker who knows nothing goes all the way and collects the +4 at Goal. After this trip, one-step TD has learned that Goal is good, but nothing about Middle or Start: the news needs a second trip to reach Middle and a third to reach Start. In a maze with a hundred steps between the start and the goal, the news travels a hundred times more slowly. This is the credit-assignment problem of Chapter 1 in its sharpest form: the reward arrives at the end, and the decisions that earned it lie far back in time.

This chapter fills the space between the two extremes. An *n-step method* uses n real rewards and then bootstraps. The *λ-return* averages all the n-step returns with geometrically decreasing weights. *Eligibility traces* compute that average online, step by step, by keeping a fading memory of the states the agent has visited; when a surprise occurs, every remembered state shares the credit or the blame. The same ideas, in modern dress, sit inside most deep reinforcement-

learning agents: n-step targets in Rainbow, generalized advantage estimation in PPO, V-trace in IMPALA and Retrace in off-policy actor-critic methods.

Section 7.1 introduces n-step returns and n-step TD and runs the classic experiment on a random walk with 19 states. Section 7.2 turns n-step TD into a control method, and Section 7.3 shows how to learn from data generated by a different policy, with importance sampling or with the tree-backup algorithm, which needs no ratios. Section 7.4 defines the λ-return, and Section 7.5 introduces the eligibility traces that compute it backward. Section 7.9 connects these ideas to deep reinforcement learning and collects practical advice. Three sections go further: Section 7.6 (●●) proves that the forward and backward views agree and derives true online TD(λ), Section 7.7 (●●) extends traces to control, and Section 7.8 (●●●) analyzes off-policy traces with guarantees. On a first reading, read Section 7.1 to 7.5 and 7.9.

7.1 Between Monte Carlo and TD: n-step returns

Throughout the chapter an episode is written $s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_T$: the reward r_t arrives right after the action a_t is taken in the state s_t, and T is the time at which the episode reaches a terminal state. The *return* from time t is $G_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_{T-1}$, where $\gamma \in [0, 1]$ is the discount factor. For now we evaluate a fixed policy π: we want a table V with one number per state such that $V(s) \approx V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$. Terminal states have value 0 and are never updated.

7.1.1 The n-step return

The Monte Carlo target for $V(s_t)$ is G_t: every reward up to the end. The TD(0) target (Algorithm 6.1) is $r_t + \gamma V(s_{t+1})$: one real reward, then an estimate that stands in for everything after it. The natural compromise is to take a few real rewards and then the estimate.

Definition 7.1 *n-step return*

For $n \geq 1$, the *n-step return* from time t is

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}), \tag{7.1}$$

where V is the current value estimate. If the episode ends first ($t + n \geq T$), the sum stops at the last reward r_{T-1} and there is nothing to bootstrap from, so $G_{t:t+n} = G_t$.

In words: follow n steps of real experience, then trust the estimate for the rest. The subscript $t{:}t{+}n$ says that the return starts at time t and uses information up to time $t + n$. With $n = 1$ it is the TD target; with $n \geq T - t$ it is the Monte Carlo return. Figure 7.1 shows the family as *backup diagrams*. Each column is the chain of states (open circles) and actions (dots) whose rewards enter the update of $V(s_t)$; the column ends at the state whose estimate closes the target, or at a terminal state (square) for Monte Carlo.

One detail matters in practice. If an episode ends because of a time limit rather than a real terminal state, then s_T still has a value, and the last returns of the episode should bootstrap from $V(s_T)$ instead of stopping (Section 6.8). Writing every n-step return as "rewards, then $\gamma^{h-t} V(s_h)$ with $h = \min(t + n, T)$" handles both cases at once, because V is 0 at a true terminal state.

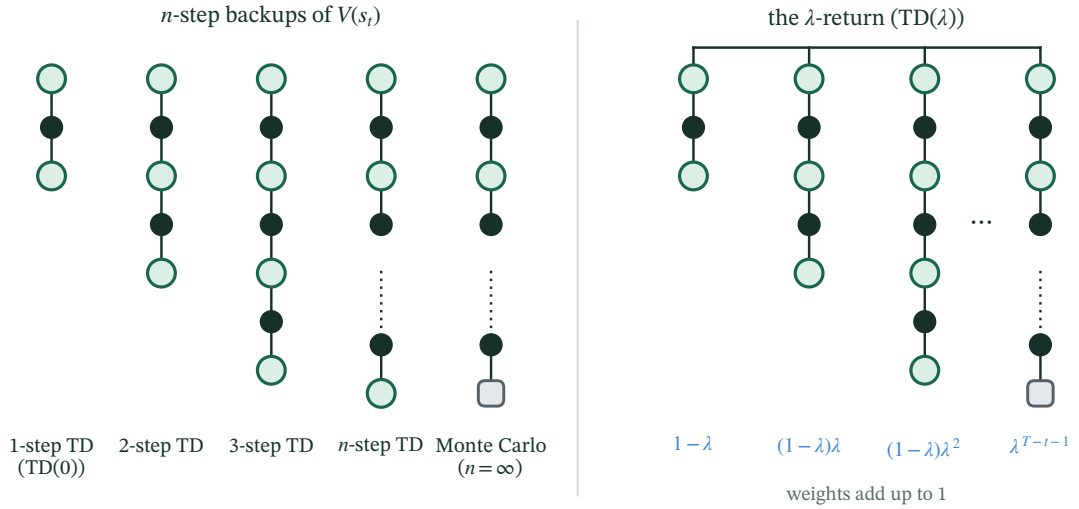


Figure 7.1. Backup diagrams. Left: the n-step backups of $V(s_t)$, from one-step TD to Monte Carlo; each column uses the rewards along its chain and the estimate at its last state. Right: the λ-return of Section 7.4 combines all of them, giving weight $(1-\lambda)\lambda^{n-1}$ to the n-step backup and the remaining weight to the complete return.

7.1.2 n-step TD

The *n-step TD* update moves $V(s_t)$ toward its n-step return:

$$V(s_t) \leftarrow V(s_t) + \eta\left[G_{t:t+n} - V(s_t)\right], \tag{7.2}$$

where $\eta \in (0, 1]$ is the step size. It is the error-correction rule of (2.2) again, with a new target. The only complication is timing. The target $G_{t:t+n}$ is known only when s_{t+n} has been observed, n steps after the visit to s_t. So the agent updates the state it visited $n-1$ steps ago: during the first $n-1$ steps of an episode it makes no updates, and after the episode ends it catches up on the last states, whose returns stop at the end.

Algorithm 7.1 n-step TD for estimating V^π

Input: a policy π to evaluate, a number of steps $n \geq 1$, step size $\eta \in (0, 1]$, discount γ.

1. Initialize $V(s)$ arbitrarily for all s, with $V(\text{terminal}) = 0$.
2. For each episode:
    1. Observe the initial state s_0; set $T \leftarrow \infty$.
    2. For $t = 0, 1, 2, \ldots$:
        1. If $t < T$: take $a_t \sim \pi(\cdot \mid s_t)$; observe r_t and s_{t+1}; if s_{t+1} is terminal, $T \leftarrow t+1$.
        2. $\tau \leftarrow t - n + 1$ (the time whose estimate is updated now).
        3. If $\tau \geq 0$:
            1. $G \leftarrow \sum_{k=\tau}^{\min(\tau+n,T)-1} \gamma^{k-\tau} r_k$.
            2. If $\tau + n < T$: $G \leftarrow G + \gamma^n V(s_{\tau+n})$.

3. $V(s_\tau) \leftarrow V(s_\tau) + \eta\,[G - V(s_\tau)]$.
4. If $\tau = T - 1$: stop; the episode is over.

Output: $V \approx V^\pi$.

The variable τ (tau) is the time whose state is being updated. The agent has to remember only the last $n + 1$ states and n rewards, and with a running sum of the discounted rewards each step costs a constant amount of work. With $n = 1$ the algorithm is TD(0); with $n \geq T$ it is constant-step-size Monte Carlo, which updates every state once the episode is over.

Example 7.1 How far back does one episode reach?

Start with $V = 0$ everywhere, $\gamma = 0.9$ and $\eta = 0.5$, and suppose the hiker's first episode goes all the way: Start, go, Middle, go, Goal, finish, with rewards $r_0 = 0$, $r_1 = 0$, $r_2 = 4$, so $T = 3$.

One-step TD ($n = 1$). The targets of $V(\text{Start})$ and $V(\text{Middle})$ are $0 + 0.9 \cdot 0 = 0$, so nothing changes there. $V(\text{Goal})$ moves halfway to 4, to 2.

Two-step TD ($n = 2$). The update of $V(\text{Start})$, made at time 1, uses $r_0 + \gamma r_1 + \gamma^2 V(\text{Goal}) = 0.81 \cdot 0 = 0$, because $V(\text{Goal})$ is still 0 when it is used. The update of $V(\text{Middle})$ uses $r_1 + \gamma r_2 = 3.6$ (the episode ends within two steps), so $V(\text{Middle}) = 1.8$. As before, $V(\text{Goal}) = 2$.

Monte Carlo ($n = 3$). Every state moves halfway to its complete return: $V(\text{Start}) = 0.5 \cdot 3.24 = 1.62$, $V(\text{Middle}) = 1.8$, $V(\text{Goal}) = 2$.

The table also shows the values after a second, identical episode:

Table 7.1. Values ($V(\text{Start})$, $V(\text{Middle})$, $V(\text{Goal})$) after one and two episodes that go all the way. The true values of this policy are 3.24, 3.6 and 4.

Method	after episode 1	after episode 2
$n = 1$ (TD(0))	(0, 0, 2)	(0, 0.9, 3)
$n = 2$	(0, 1.8, 2)	(0.81, 2.7, 3)
$n = 3$ (Monte Carlo)	(1.62, 1.8, 2)	(2.43, 2.7, 3)

One-step TD carries the news of the +4 back by one state per episode; n-step TD carries it back by n states.

♦

7.1.3 Why looking further ahead helps

The n-step return replaces the tail of the return with an estimate. If the estimate is wrong, the target is wrong, but only by a discounted amount, because n real rewards come first. This is called the *error-reduction property*.

Proposition 7.2 Error reduction

For every value table V, every state s and every $n \geq 1$,

$$\left|\mathbb{E}_\pi[G_{t:t+n} \mid s_t = s] - V^\pi(s)\right| \leq \gamma^n \max_{s'} |V(s') - V^\pi(s')|. \tag{7.3}$$

Proof. Let $\mathcal{T}^\pi$ be the Bellman expectation operator of Section 3.6, $(\mathcal{T}^\pi V)(s) = \sum_a \pi(a \mid s)\big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')\big]$. Conditioning on the first step gives $\mathbb{E}_\pi[G_{t:t+1} \mid s_t = s] = (\mathcal{T}^\pi V)(s)$. Since $G_{t:t+n} = r_t + \gamma G_{t+1:t+n}$, where $G_{t+1:t+n}$ is the $(n-1)$-step return from $t+1$, induction on n and the Markov property give $\mathbb{E}_\pi[G_{t:t+n} \mid s_t = s] = ((\mathcal{T}^\pi)^n V)(s)$. The true values are a fixed point, $V^\pi = (\mathcal{T}^\pi)^n V^\pi$, so the difference is $(\mathcal{T}^\pi)^n V - (\mathcal{T}^\pi)^n V^\pi = \gamma^n (P^\pi)^n (V - V^\pi)$, where P^π is the state-transition matrix of π ((3.9)). Each row of $(P^\pi)^n$ is a probability distribution, so each entry of $(P^\pi)^n(V - V^\pi)$ is an average of entries of $V - V^\pi$ and is at most $\max_{s'} |V(s') - V^\pi(s')|$ in absolute value. ■

In episodic problems the bound is even better than it looks: terminal states have zero error, so the factor γ^n can be multiplied by the probability that the episode lasts at least n more steps. As a check, take the Trail with the uniform policy, whose values are $V^\pi = (1.76, 2.8, 4)$, and the initial guess $V = 0$, whose largest error is 4. The expected one-step targets are $(0.5, 1, 4)$, with errors $(1.26, 1.8, 0)$, within the bound $0.9 \times 4 = 3.6$. The expected two-step targets are $(0.95, 2.8, 4)$, with errors $(0.81, 0, 0)$, and the three-step targets are exact.

The proposition says that the bias that bootstrapping introduces shrinks like γ^n. The price is variance: the target now contains n random rewards and n random transitions. This is the bias–variance trade-off of Section 6.2 made continuous, with n as the dial. The proposition also explains why n-step TD works at all: its expected update moves V toward $(\mathcal{T}^\pi)^n V$, and $(\mathcal{T}^\pi)^n$ is a γ^n-contraction with fixed point V^π (Theorem 4.3). With step sizes that decrease in the usual way, tabular n-step TD converges to V^π (Sutton and Barto 2018, sec. 7.1).

7.1.4 The random walk with 19 states

Which n is best? The classic test problem is a random walk with 19 states in a row (`RandomWalk(19)` in `rlcourse.envs`). Every episode starts in the middle, state 10. At each step the walker moves one state left or right with probability one half each. Leaving the row on the right earns +1; leaving it on the left earns 0; every other step earns 0, and $\gamma = 1$. The true value of state i is $i/20$, the probability of leaving on the right, and an episode lasts 100 steps on average.

The experiment starts every estimate at 0.5, halfway between the two possible outcomes. For a given n and step size η, it runs 10 episodes, measures the root-mean-square (RMS) error of the 19 estimates at the end of each episode, and averages the error over the 10 episodes and over 100 independent runs. The initial guess has RMS error $\sqrt{30}/20 \approx 0.274$.

Figure 7.2 shows the result. Each curve is U-shaped: a tiny step size learns too little in 10 episodes, and a large one makes the estimates chase the noise of individual episodes. The best results come from intermediate n. Four-step TD reaches an average error of 0.133 (at $\eta \approx 0.375$),

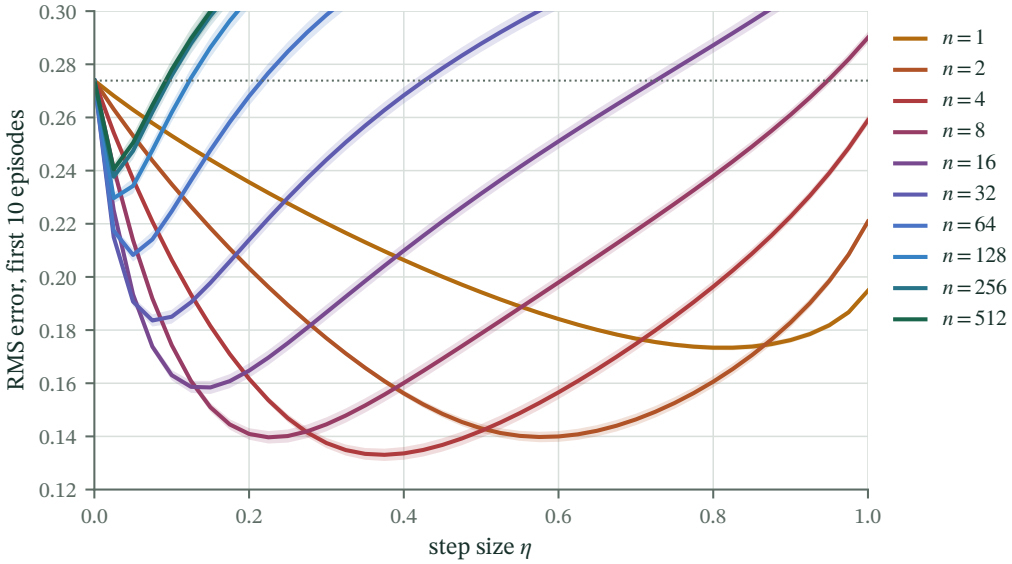


Figure 7.2. n-step TD on the random walk with 19 states: RMS error of the estimates, averaged over the first 10 episodes and 100 runs, as a function of the step size η, for n from 1 to 512. The shaded bands are ± 1 standard error over the runs, and the dotted line is the error of the initial guess. An intermediate n does best, and large n needs small step sizes.

while one-step TD gets no lower than 0.173 (at $\eta \approx 0.8$) and 512-step TD, which is Monte Carlo for almost every episode, no lower than 0.240 (at $\eta \approx 0.025$).

The reasons are the two weaknesses from the introduction. With only 10 episodes, one-step TD has not had time to carry the information from the ends of the row to its middle. Monte Carlo moves information quickly, but every one of its targets is a single noisy outcome, 0 or 1, so it needs a small step size. A few steps of lookahead get most of the speed of Monte Carlo with much of the stability of TD. The best n also depends on how much experience there is: after 100 episodes, when the estimates have become good enough to bootstrap from, the lowest final error comes from $n = 1$ (Exercise 7.8).

PITFALL

Comparing two values of n at one shared step size can reverse the conclusion. In Figure 7.2, at $\eta = 0.8$ one-step TD (error 0.173) beats every $n \geq 4$, while at $\eta = 0.2$ it (error 0.236) is worse than every n from 2 to 32. The best step size shrinks as n grows, so n and η must be tuned together, and a fair comparison shows each method at its own best step size, or the whole curve.

CHECKPOINT

1. What is $G_{t:t+n}$ when $n = 1$? When $n \geq T - t$?
2. At what time does n-step TD update the estimate of s_t?
3. Why does the expected n-step target get closer to $V^\pi(s_t)$ as n grows, even if V is badly wrong?

Answers.

1. The TD(0) target $r_t + \gamma V(s_{t+1})$; the Monte Carlo return G_t.
2. At the end of step $t+n-1$, as soon as r_{t+n-1} and s_{t+n} are known (or at the end of the episode, if it ends sooner).
3. The error of the estimate enters only through $\gamma^n V(s_{t+n})$, so it is multiplied by γ^n (and by the probability that the episode is still running), as in Proposition 7.2.

7.2 n-step SARSA

For control we learn action values and act ε-greedily, as SARSA does (Algorithm 6.2). The n-step return for action values bootstraps from the action actually chosen n steps later:

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n Q(s_{t+n}, a_{t+n}), \tag{7.4}$$

again with $G_{t:t+n} = G_t$ if the episode ends before $t+n$. The update is $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta\,[G_{t:t+n} - Q(s_t,a_t)]$.

Algorithm 7.2 n-step SARSA

Input: number of steps $n \geq 1$, step size $\eta \in (0,1]$, exploration rate $\varepsilon > 0$, discount γ.

1. Initialize $Q(s,a)$ arbitrarily for all s, a, with $Q(\text{terminal},\cdot) = 0$.
2. For each episode:
   1. Observe s_0; choose a_0 ε-greedily with respect to Q; set $T \leftarrow \infty$.
   2. For $t = 0, 1, 2, \ldots$:
      1. If $t < T$: take a_t; observe r_t and s_{t+1}; if s_{t+1} is terminal, $T \leftarrow t+1$; else choose a_{t+1} ε-greedily with respect to Q.
      2. $\tau \leftarrow t - n + 1$.
      3. If $\tau \geq 0$:
         1. $G \leftarrow \sum_{k=\tau}^{\min(\tau+n,T)-1} \gamma^{k-\tau} r_k$.
         2. If $\tau + n < T$: $G \leftarrow G + \gamma^n Q(s_{\tau+n}, a_{\tau+n})$.
         3. $Q(s_\tau,a_\tau) \leftarrow Q(s_\tau,a_\tau) + \eta\,[G - Q(s_\tau,a_\tau)]$.
      4. If $\tau = T-1$: stop; the episode is over.

Output: Q, and the ε-greedy policy with respect to it.

A useful variant replaces the last term of (7.4) by its expectation under the current policy, $\gamma^n \bar{V}(s_{t+n})$ with $\bar{V}(s) = \sum_a \pi(a \mid s) Q(s,a)$. This is *$n$-step Expected SARSA* (compare Algorithm 6.4). It removes the randomness of the last action, and it is the natural starting point for the off-policy methods of the next section.

Example 7.2 n-step SARSA in a maze

The maze, which reappears in Chapter 8, is a grid of 6 rows and 9 columns with three walls, a start on the left and a goal in the top-right corner. In the lab it is written as a list of strings:

```
MAZE = ['.......#G',
        '..#....#.',
        'S.#....#.',
        '..#......',
        '.....#...',
        '.........']
```

Moves into a wall or off the grid leave the agent in place, reaching G earns +1 and ends the episode, and every other step earns 0. We use $\gamma = 0.95$, $\varepsilon = 0.1$, $\eta = 0.3$ and $Q = 0$ at the start.

Because all estimates start equal, the first episode is a random walk through the maze, about 1,200 steps long on average (over 20 seeds). When it ends, one-step SARSA has changed exactly one action value: the one for the final move into the goal, the only update whose target was not zero. Four-step SARSA has changed the values of the last four moves, and 16-step SARSA those of the last 16 (on average 3.9 and 12.4 distinct pairs, because a wandering path repeats some pairs). In its second episode, one-step SARSA must wander until it stumbles on its single informed pair, while n-step SARSA finds a short trail of informed pairs. Over the first 30 episodes, the agents need on average 5,084 steps in total for $n = 1$, 2,727 for $n = 4$ and 2,333 for $n = 16$. About 1,200 steps of each total are the uninformed first episode, which is the same for every n.

7.3 Off-policy n-step learning

Q-learning (Algorithm 6.3) learns about the greedy policy while the agent explores, and it needs no correction for doing so. The reason is that its target bootstraps immediately: the only action whose consequences appear in the target is a_t, the action being evaluated, and $Q(s_t, a_t)$ is the value of taking it no matter who chose it. With $n > 1$ this is no longer true. The rewards $r_{t+1}, \dots, r_{t+n-1}$ follow actions $a_{t+1}, \dots, a_{t+n-1}$ chosen by the *behavior policy* b, and if b takes an action that the *target policy* π would never take, the rewards after it describe b, not π. There are two ways to repair the target: reweight it, or stop sampling where the policies disagree.

7.3.1 Importance-sampling corrections

The first repair is the importance-sampling idea of Section 5.4. The *importance-sampling ratio* of the action at time k is $\rho_k = \pi(a_k \mid s_k)/b(a_k \mid s_k)$, and for a stretch of time we write

$$\rho_{t:h} = \prod_{k=t}^{\min(h,\,T-1)} \rho_k .$$

Off-policy n-step TD for V^π weights each update by the ratios of the n actions whose consequences its return contains:

$$V(s_t) \leftarrow V(s_t) + \eta\, \rho_{t:t+n-1} \big[G_{t:t+n} - V(s_t) \big]. \tag{7.5}$$

For action values, off-policy n-step SARSA uses the return (7.4) and the update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\, \rho_{t+1:t+n} \big[G_{t:t+n} - Q(s_t, a_t) \big]. \tag{7.6}$$

Notice which ratios appear. The first action a_t needs no correction, because $Q(s_t, a_t)$ is the value of having taken it. The last action a_{t+n} needs one, because the bootstrap term $Q(s_{t+n}, a_{t+n})$ depends on it. For n-step Expected SARSA, which bootstraps from $\bar{V}(s_{t+n})$, the product stops at $t+n-1$. When $\pi = b$ every ratio is 1 and the on-policy algorithms come back. The weighted target is right on average: $\mathbb{E}_b[\rho_{t:t+n-1} G_{t:t+n} \mid s_t = s] = \mathbb{E}_\pi[G_{t:t+n} \mid s_t = s]$ for any fixed V, by the same reweighting argument as in (5.4) (Exercise 7.13).

Example 7.3 Off-policy targets on the Trail

The target policy π always goes; the behavior policy b is uniform. Suppose the estimates downstream are already correct for π, $Q(\text{Middle}, \text{go}) = 3.6$ and $Q(\text{Goal}, \text{finish}) = 4$, while $Q(\text{Start}, \text{go}) = 2.52$, its value under the uniform policy. We update the pair (Start, go) with $n = 2$, $\gamma = 0.9$ and $\eta = 0.1$. Under b, the episodes that start with go come in two equally likely kinds.

Episode A: Start, go, Middle, go, Goal, finish, with rewards $0, 0, 4$. The return bootstraps from Goal: $G_{0:2} = 0 + 0.9 \cdot 0 + 0.81 \cdot 4 = 3.24$. The ratio covers $a_1 = \text{go}$, with $\pi/b = 1/0.5 = 2$, and $a_2 = \text{finish}$, the only action in Goal, with ratio 1; so $\rho_{1:2} = 2$ and $Q(\text{Start}, \text{go})$ becomes $2.52 + 0.1 \cdot 2 \cdot (3.24 - 2.52) = 2.664$.

Episode B: Start, go, Middle, exit, with rewards $0, 2$. The target policy never exits, so $\rho_{1:2} = \rho_1 = 0$ (the episode ends at $T = 2$, so the product stops at a_1), and nothing changes.

On average the change is $\frac{1}{2} \cdot 0.144 + \frac{1}{2} \cdot 0 = 0.072 = 0.1 \cdot (3.24 - 2.52)$, exactly the change of an update toward the true value $Q^\pi(\text{Start}, \text{go}) = 3.24$. But each individual update is either doubled or thrown away.

♦

The example shows the weakness. Each ratio can be larger than one, and a product of n of them can be very large, so a few updates are huge and many are zero. Longer returns make this worse, exactly as for Monte Carlo importance sampling (Section 5.5). Per-decision ratios with control variates (Section 5.5.3) reduce the variance but do not remove the problem.

7.3.2 The tree-backup algorithm

The second repair needs no ratios at all. One-step Expected SARSA needs none because it averages over the next action with the target policy's probabilities instead of sampling it. *Tree backup* (Precup et al. 2000) extends this to n steps. At every state along the way, the actions that were *not* taken contribute their current estimates, weighted by their probabilities under π, and the action that *was* taken contributes the rest of the sampled return, weighted by its probability under π. The backup diagram is a spine (the sampled path) with branches (the untaken actions) at every state, hence the name.

The tree-backup return is built backward from the end of its n steps. Its last step is the Expected SARSA target, $G_{t+n-1:t+n} = r_{t+n-1} + \gamma \sum_a \pi(a \mid s_{t+n})\, Q(s_{t+n}, a)$, or simply r_{T-1} if the episode

ends first. Each earlier step $k = t + n - 2, \dots, t$ adds

$$G_{k:t+n} = r_k + \gamma \sum_{a \neq a_{k+1}} \pi(a \mid s_{k+1})\, Q(s_{k+1}, a) + \gamma\, \pi(a_{k+1} \mid s_{k+1})\, G_{k+1:t+n}. \tag{7.7}$$

The update is $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\,[G_{t:t+n} - Q(s_t, a_t)]$, with no ratio.

Algorithm 7.3 *n*-step tree backup

Input: a target policy π, any behavior policy b, number of steps $n \geq 1$, step size $\eta \in (0, 1]$, discount γ.

1. Initialize $Q(s, a)$ arbitrarily for all s, a, with $Q(\text{terminal}, \cdot) = 0$.
2. For each episode:
    1. Observe s_0; choose $a_0 \sim b(\cdot \mid s_0)$; set $T \leftarrow \infty$.
    2. For $t = 0, 1, 2, \dots$:
        1. If $t < T$: take a_t; observe r_t and s_{t+1}; if s_{t+1} is terminal, $T \leftarrow t+1$; else choose $a_{t+1} \sim b(\cdot \mid s_{t+1})$.
        2. $\tau \leftarrow t - n + 1$.
        3. If $\tau \geq 0$:
            1. $h \leftarrow \min(\tau + n, T)$.
            2. If $h = T$: $G \leftarrow r_{T-1}$; else $G \leftarrow r_{h-1} + \gamma \sum_a \pi(a \mid s_h)\, Q(s_h, a)$.
            3. For $k = h - 2$ down to τ: $G \leftarrow r_k + \gamma \sum_{a \neq a_{k+1}} \pi(a \mid s_{k+1}) Q(s_{k+1}, a) + \gamma\, \pi(a_{k+1} \mid s_{k+1})\, G$.
            4. $Q(s_\tau, a_\tau) \leftarrow Q(s_\tau, a_\tau) + \eta\,[G - Q(s_\tau, a_\tau)]$.
        4. If $\tau = T - 1$: stop; the episode is over.

Output: $Q \approx Q^\pi$.

In the tree-backup return the sampled continuation after k steps is weighted by $\pi(a_{t+1} \mid s_{t+1}) \cdots \pi(a_{t+k} \mid s_{t+k})$, a product of probabilities, never larger than one. Nothing is multiplied by a large ratio, so the variance stays small. The price shows when π is greedy: the weight drops to zero at the first exploratory action of b, and the return becomes short. Tree backup looks far ahead only while the behavior does what the target policy would do. Even on-policy, with a stochastic $\pi = b$, it shrinks the sampled continuation at every step, which makes it slower than plain n-step SARSA there.

Return to Example 7.3. For episode A, the branch term vanishes because $\pi(\text{exit} \mid \text{Middle}) = 0$, and $G_{0:2} = 0 + 0.9 \cdot 1 \cdot (0 + 0.9 \cdot 4) = 3.24$. For episode B, the action taken has $\pi(\text{exit} \mid \text{Middle}) = 0$, so its continuation is dropped and the untaken branch supplies the estimate: $G_{0:2} = 0 + 0.9 \cdot 1 \cdot Q(\text{Middle}, \text{go}) = 3.24$. Both episodes move $Q(\text{Start}, \text{go})$ to $2.52 + 0.1 \cdot 0.72 = 2.592$: the same average change as importance sampling, with no variance at all, because the downstream estimates are already right. With noisier data the difference persists. In Lab 7.1, 3-step updates for the target that always goes, on the slippery Trail and from 400 episodes of uniform behavior ($\eta = 0.3$, 20 seeds) leave an RMS error of 0.102 on average and 0.166 at worst with importance sampling, against 0.071 and 0.111 with tree backup.

7.3.3 A unifying algorithm: Q(σ)

Importance sampling samples every action and corrects with ratios; tree backup takes the expectation over every action and never samples further than π allows. *Q(σ)* (De Asis et al. 2018; Sutton and Barto 2018, sec. 7.6) chooses between the two at every step with a parameter $\sigma_k \in [0,1]$: $\sigma_k = 1$ samples, $\sigma_k = 0$ takes the expectation. With $\bar{V}(s) = \sum_a \pi(a \mid s) Q(s,a)$ and the horizon $h = t + n$, the return is built backward from $G_{h:h} = Q(s_h, a_h)$ (or from $G_{T-1:T} = r_{T-1}$ if the episode ends first) by

$$G_{k:h} = r_k + \gamma\big[\sigma_{k+1}\rho_{k+1} + (1-\sigma_{k+1})\,\pi(a_{k+1} \mid s_{k+1})\big]\big(G_{k+1:h} - Q(s_{k+1}, a_{k+1})\big) + \gamma\,\bar{V}(s_{k+1}). \quad (7.8)$$

With $\sigma \equiv 0$ this is the tree-backup return, as you can check by writing $\sum_{a \neq a_{k+1}} \pi(a \mid s) Q(s,a) = \bar{V}(s) - \pi(a_{k+1} \mid s) Q(s, a_{k+1})$ in (7.7). With $\sigma \equiv 1$ it is n-step SARSA with per-decision importance sampling, written with a control variate: the term $\gamma \bar{V}(s_{k+1}) - \gamma\rho_{k+1} Q(s_{k+1}, a_{k+1})$ has mean zero and reduces the variance. De Asis et al. (2018) found intermediate and time-varying values of σ to work well in their experiments.

7.4 The λ-return

No single n is best for every problem, and the best n changes as learning proceeds. A natural response is to average. Any weighted average of n-step returns whose weights add up to one is a sensible target with the same expected fixed point, for example $\frac{1}{2} G_{t:t+2} + \frac{1}{2} G_{t:t+4}$; such averages are called *compound returns*. The λ-return is the average with geometric weights. It has a single parameter, and, as Section 7.5 shows, it can be computed incrementally.

Definition 7.3 λ-return

For $\lambda \in [0,1]$, the *λ-return* from time t in an episode that ends at time T is

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1}\, G_{t:t+n} + \lambda^{T-t-1}\, G_t. \quad (7.9)$$

In a continuing task it is $G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$.

The one-step return gets weight $1-\lambda$, the two-step return $(1-\lambda)\lambda$, and so on; every n-step return that reaches the end of the episode equals G_t, so all the remaining weight, λ^{T-t-1}, lands on the complete return (Figure 7.3). The weights add up to one (Exercise 7.2). Two special cases anchor the family: with $\lambda = 0$, $G_t^0 = G_{t:t+1}$ is the TD target; with $\lambda = 1$, $G_t^1 = G_t$ is the Monte Carlo return. In a continuing task the weights $(1-\lambda)\lambda^{n-1}$ have mean $1/(1-\lambda)$, so $\lambda = 0.8$ averages n-step returns around a lookahead of 5 steps, and $\lambda = 0.9$ around 10.

For a fixed table V, the λ-returns of an episode satisfy a simple recursion:

$$G_t^\lambda = r_t + \gamma\big[(1-\lambda)\, V(s_{t+1}) + \lambda\, G_{t+1}^\lambda\big], \quad (7.10)$$

started from $G_T^\lambda = V(s_T)$, which is 0 at a terminal state. In words: after each real reward, flip a biased coin; with probability $1-\lambda$ stop and bootstrap from the estimate of the next state, and

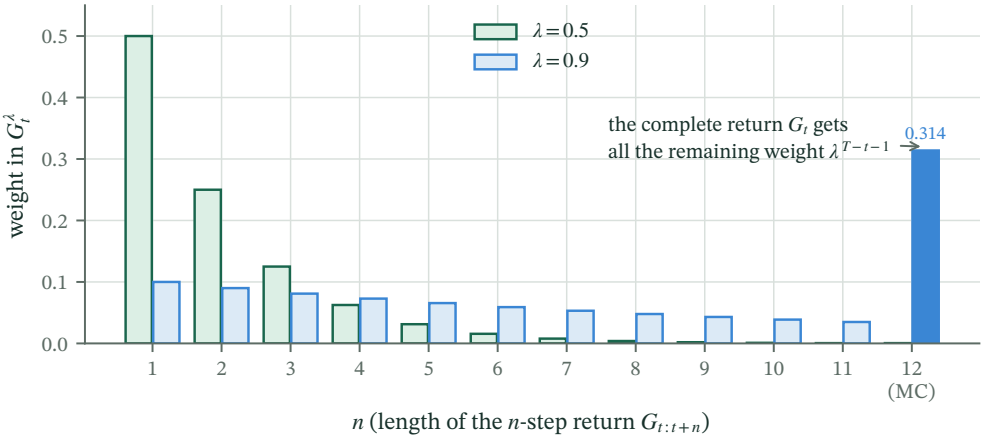


Figure 7.3. The weights of the n-step returns in the λ-return of a state whose episode ends 12 steps later. With $\lambda = 0.5$ almost all the weight is on short returns; with $\lambda = 0.9$ the weights decay slowly, and the complete (Monte Carlo) return, the last bar, receives all the remaining weight, $0.9^{11} \approx 0.314$.

with probability λ go on and use the next λ-return. Lemma 7.5 derives the recursion from (7.9). It gives all the λ-returns of an episode in a single backward pass.

The *forward view* of learning with λ-returns updates each visited state toward its λ-return. Since G_t^λ depends on the whole rest of the episode, the updates have to wait until the episode is over. The *offline λ-return algorithm* holds V fixed during the episode; at the end it computes every G_t^λ with this V and changes each visited state by $\eta\,[G_t^\lambda - V(s_t)]$, adding up the changes of a state that was visited more than once.

Example 7.4 A λ-return on the Trail

Take again the episode Start, go, Middle, go, Goal, finish with rewards $0, 0, 4$, the table $V = 0$, $\gamma = 0.9$ and $\lambda = 0.5$. The recursion runs backward from $G_3^\lambda = 0$:

$$G_2^\lambda = 4, \qquad G_1^\lambda = 0 + 0.9\,(0.5 \cdot 0 + 0.5 \cdot 4) = 1.8, \qquad G_0^\lambda = 0 + 0.9\,(0.5 \cdot 0 + 0.5 \cdot 1.8) = 0.81.$$

The definition gives the same for $t = 0$: the n-step returns are $G_{0:1} = 0$, $G_{0:2} = 0$ and $G_0 = 3.24$, so $G_0^\lambda = 0.5\,(0 + 0.5 \cdot 0) + 0.25 \cdot 3.24 = 0.81$. With $\eta = 0.5$, the offline λ-return algorithm changes the values to $(0.405, 0.9, 2)$. That lies between one-step TD, $(0, 0, 2)$, and Monte Carlo, $(1.62, 1.8, 2)$, in Table 7.1: every state learns something, by amounts that shrink by $\gamma\lambda = 0.45$ per step back from Goal.

♦

The forward view is a clean definition but an awkward algorithm. Nothing is learned during an episode, the whole episode must be stored, and in a continuing task the λ-return is never available. The backward view of the next section removes all three problems.

7.5 Eligibility traces and TD(λ)

The key to the backward view is an identity, proved as Lemma 7.5: for a fixed table V,

$$G_t^\lambda - V(s_t) = \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t}\,\delta_k, \qquad \delta_k = r_k + \gamma V(s_{k+1}) - V(s_k). \tag{7.11}$$

The error of the forward view at time t is a sum of the TD errors ((6.1)) that follow time t, each discounted by $\gamma\lambda$ per step of delay. Now read the sum the other way around. The TD error at time k contributes $(\gamma\lambda)^{k-t}$ to the update of every earlier visit at time $t \le k$. So instead of waiting for future TD errors, the agent can hand out each TD error, as soon as it happens, to all the states visited before, in proportion to $(\gamma\lambda)^{\text{time since the visit}}$. The bookkeeping needs one number per state.

Definition 7.4 Eligibility trace

The *accumulating eligibility trace* of state s starts each episode at $z_{-1}(s) = 0$ and is updated at every step by

$$z_t(s) = \gamma\lambda\, z_{t-1}(s) + \mathbb{1}\{s_t = s\}. \tag{7.12}$$

Unrolling the recursion, $z_t(s) = \sum_{k \le t,\ s_k = s} (\gamma\lambda)^{t-k}$: each visit to s so far counts one, faded by $\gamma\lambda$ for every step since it happened. The algorithm *TD(λ)* computes the TD error after each step and moves every state in proportion to its trace:

$$V(s) \leftarrow V(s) + \eta\,\delta_t\, z_t(s) \qquad \text{for every state } s. \tag{7.13}$$

Algorithm 7.4 TD(λ) with eligibility traces

Input: a policy π to evaluate, trace decay $\lambda \in [0,1]$, step size $\eta \in (0,1]$, discount γ.

1. Initialize $V(s)$ arbitrarily for all s, with $V(\text{terminal}) = 0$.
2. For each episode:
   1. Observe s_0; set $z(s) \leftarrow 0$ for all s.
   2. For $t = 0, 1, 2, \ldots$ until s_t is terminal:
      1. Take $a_t \sim \pi(\cdot \mid s_t)$; observe r_t and s_{t+1}.
      2. $\delta \leftarrow r_t + \gamma V(s_{t+1}) - V(s_t)$.
      3. $z(s) \leftarrow \gamma\lambda\, z(s)$ for all s; then $z(s_t) \leftarrow z(s_t) + 1$ (accumulating) or $z(s_t) \leftarrow 1$ (replacing).
      4. $V(s) \leftarrow V(s) + \eta\,\delta\, z(s)$ for all s.

Output: $V \approx V^\pi$.

The trace is a short-term memory of which states are *eligible* for credit or blame. A TD error is a surprise, good or bad, and the states that led to it most recently get the largest share. The two ends of the family are familiar. With $\lambda = 0$, the trace is 1 for the current state and 0 for

every other, so TD(0) comes back. With $\lambda = 1$ and $\gamma = 1$, traces never fade, every visited state receives every later TD error, and in the offline case the TD errors from time t to the end add up to $G_t - V(s_t)$, so TD(1) makes the Monte Carlo update. It makes it step by step, however, so TD(1) is a form of Monte Carlo that learns during the episode and also works in continuing tasks.

Do the forward view and the backward view really agree? If V is held fixed during the episode and the changes are applied at the end (*offline* TD(λ)), they agree exactly: by (7.11), offline TD(λ) and the offline λ-return algorithm change every state by the same total amount (Theorem 7.6, proved in Section 7.6). The online algorithm, which changes V after every step, differs slightly, by terms of order η^2, and in exchange it learns during the episode.

TD(λ) costs one trace per state. In a table only the recently visited states have traces worth keeping, since a trace that has faded below a small threshold can be set to zero. With linear function approximation, $V_w(s) = w^\top \phi(s)$, the trace becomes a vector, $z_t = \gamma\lambda z_{t-1} + \phi(s_t)$, and the update becomes $w \leftarrow w + \eta\,\delta_t z_t$, at the same cost per step as semi-gradient TD(0) (Algorithm 9.2).

Example 7.5 Traces when a state repeats

On the slippery Trail a go fails with probability 0.2 and leaves the hiker where she is. Take the episode Start (the go fails), Start (go), Middle (exit, +2): so $s_0 = s_1 =$ Start, $s_2 =$ Middle, with rewards 0, 0, 2. Use $\gamma = 0.9$, $\lambda = 0.5$ (so $\gamma\lambda = 0.45$), $\eta = 0.5$ and offline updates from $V = 0$.

The TD errors are $\delta_0 = 0$, $\delta_1 = 0$ and $\delta_2 = 2$. At time 2 the traces are $z_2(\text{Start}) = 0.45^2 + 0.45 = 0.6525$ (one term per visit) and $z_2(\text{Middle}) = 1$. The updates are $V(\text{Start}) = 0.5 \cdot 2 \cdot 0.6525 = 0.6525$ and $V(\text{Middle}) = 0.5 \cdot 2 \cdot 1 = 1$.

The forward view agrees. By (7.10), the two visits to Start have λ-returns $G_1^\lambda = 0.9\,(0.5 \cdot 0 + 0.5 \cdot 2) = 0.9$ and $G_0^\lambda = 0.9\,(0 + 0.5 \cdot 0.9) = 0.405$, so the offline λ-return algorithm adds $0.5 \cdot 0.405 + 0.5 \cdot 0.9 = 0.6525$ to $V(\text{Start})$, and $0.5 \cdot 2 = 1$ to $V(\text{Middle})$.

♦

7.5.1 Accumulating, replacing and dutch traces

An accumulating trace adds one at each visit, so a state visited several times in quick succession can have a trace well above one (Figure 7.4). With a large step size the effective step for that state, $\eta z(s)$, can then exceed one, and its estimate overshoots. Two variants avoid this. A *replacing trace* (Singh and Sutton 1996) is reset to one at a visit instead of being incremented: $z_t(s_t) = 1$, while the other traces fade as before. In Example 7.5 it would give $V(\text{Start}) = 0.5 \cdot 2 \cdot 0.45 = 0.45 = \eta\, G_1^\lambda$: credit for the most recent visit only. With $\lambda = 1$ and $\gamma = 1$, offline TD(1) with replacing traces makes the first-visit Monte Carlo update, and with accumulating traces the every-visit update (Exercise 7.10). A *dutch trace* (Seijen and Sutton 2014) sets $z_t(s_t) = (1 - \eta)\,\gamma\lambda\, z_{t-1}(s_t) + 1$. It lies between the two: as $\eta \to 0$ it becomes an accumulating trace and at $\eta = 1$ a replacing one, and it arises naturally from true online TD(λ) (Section 7.6.3).

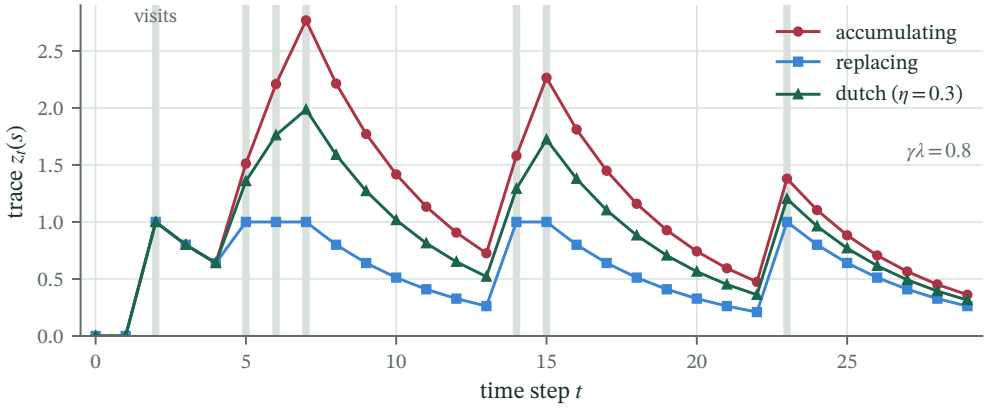


Figure 7.4. The eligibility trace of one state over 30 steps, with $\gamma\lambda = 0.8$; the gray bars mark the visits. An accumulating trace adds one at each visit and climbs to 2.77 after three visits in a row; a replacing trace is reset to one; a dutch trace with $\eta = 0.3$ lies in between. Between visits all three fade by the same factor 0.8 per step.

7.5.2 Choosing λ: the random walk again

Figure 7.5 repeats the random-walk experiment of Figure 7.2 with λ in place of n. The best λ is about 0.8 for both methods shown: TD(0.8) with accumulating traces reaches an average error of 0.133 (at $\eta \approx 0.33$), and true online TD(0.8), introduced in Section 7.6.3, reaches 0.129 (at $\eta = 0.4$). TD(0.8) with replacing traces does slightly better still, 0.125. These are the same accuracies as the best n-step method, 0.133 for $n = 4$, which fits the rule of thumb that $\lambda = 0.8$ averages around a lookahead of 5 steps.

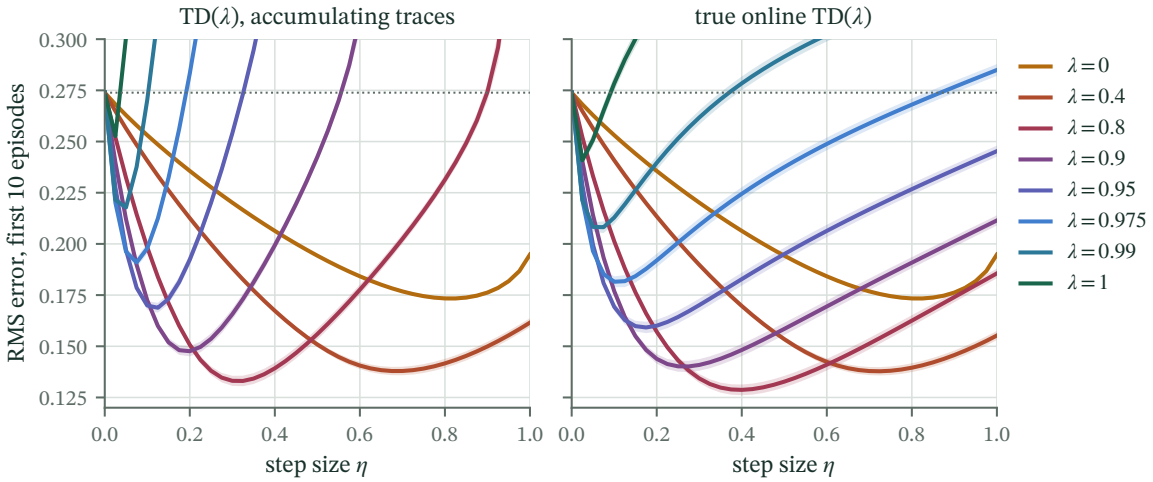


Figure 7.5. TD(λ) on the random walk with 19 states, in the setting of Figure 7.2: online TD(λ) with accumulating traces (left) and true online TD(λ) (right). Shaded bands: ± 1 standard error over 100 runs. Both reach their lowest errors near $\lambda = 0.8$. With accumulating traces, large λ and large η make the estimates blow up; true online TD(λ) stays bounded for every setting.

The two panels differ at large step sizes. With accumulating traces and $\lambda \geq 0.9$ the estimates diverge once η is large: a state that the walk visits many times in a row builds up a large trace,

and its estimate overshoots further at every step. True online TD(λ) never gets worse than an average error of 0.40 anywhere in the plotted range. The offline λ-return algorithm (not shown) does markedly worse in this experiment, 0.165 at best (with $\lambda = 0.9$ and $\eta = 0.15$), because it learns nothing until an episode ends and then adds up the changes of the many visits to each state.

LAB

Lab 7.1: Multi-step returns and eligibility traces (`labs/ch07_nstep.py`, about 45 minutes). Run `python3 labs/ch07_nstep.py`. You will reproduce Example 7.1 and Example 7.5, sweep n and η on the random walk, check numerically that the offline forward and backward views agree, watch accumulating traces blow up where true online TD(λ) does not, speed up control in the maze with n-step SARSA, and compare importance sampling with tree backup for off-policy evaluation.

TRY IT INTERACTIVELY

The interactive course in `Apps_claude/RL Course/index.html` (Module 4, lesson "n-step returns and λ") builds n-step targets on the Trail one reward at a time with a dial for n, and shows the λ-return as a weighted average of them.

CHECKPOINT

1. A state is visited at times 2 and 5. With $\gamma\lambda = 0.5$, what is its accumulating trace at time 6? Its replacing trace?
2. Which value of λ turns TD(λ) into TD(0)? Which turns offline TD(λ) into Monte Carlo?
3. Why can accumulating traces make TD(λ) unstable with a large step size?

Answers.

1. Accumulating: $0.5^4 + 0.5 = 0.5625$. Replacing: the visit at time 5 resets the trace to 1, so it is 0.5 at time 6.
2. $\lambda = 0$ gives TD(0); $\lambda = 1$ (with offline updates) gives the every-visit Monte Carlo update.
3. A state visited often in a short time gets a trace above one, so its estimate moves by $\eta z(s)\,\delta$, which can be more than the whole TD error; the estimate overshoots and the next TD error is larger still.

7.6 Forward-backward equivalence ●●

This section proves the identity (7.11) and the offline equivalence of the two views, explains what goes wrong online, and derives the true online algorithm that repairs it.

7.6.1 The offline equivalence theorem

Lemma 7.5 The λ-return error is a discounted sum of TD errors

Fix a value table V with $V(s_T) = 0$ if s_T is terminal, and set $G_T^\lambda = V(s_T)$. For every $t < T$ the

λ-return of (7.9) satisfies the recursion (7.10), and

$$G_t^\lambda - V(s_t) = \sum_{k=t}^{T-1} (\gamma\lambda)^{k-t}\,\delta_k, \qquad \delta_k = r_k + \gamma V(s_{k+1}) - V(s_k).$$

Proof. *The recursion.* Extend the n-step returns by $G_{t:t+n} = G_t$ for $n \geq T-t$. Then for $\lambda < 1$ the definition can be written $G_t^\lambda = (1-\lambda)\sum_{n\geq 1} \lambda^{n-1} G_{t:t+n}$, because the weights of all the $n \geq T-t$ add up to $(1-\lambda)\sum_{n\geq T-t}\lambda^{n-1} = \lambda^{T-t-1}$. For $t < T-1$, each n-step return splits after its first reward as $G_{t:t+n} = r_t + \gamma G_{t+1:t+n}$, with the convention $G_{t+1:t+1} = V(s_{t+1})$. Hence

$$\begin{aligned} G_t^\lambda &= r_t + \gamma(1-\lambda)\Big[V(s_{t+1}) + \sum_{n\geq 2} \lambda^{n-1} G_{t+1:t+n}\Big] \\ &= r_t + \gamma(1-\lambda)V(s_{t+1}) + \gamma\lambda \underbrace{(1-\lambda)\sum_{m\geq 1} \lambda^{m-1} G_{t+1:t+1+m}}_{= G_{t+1}^\lambda}, \end{aligned}$$

where the first line uses $(1-\lambda)\sum_{n\geq 1}\lambda^{n-1} = 1$ and the second substitutes $m = n-1$. For $t = T-1$ every n-step return equals r_{T-1}, which is what the recursion gives with $G_T^\lambda = V(s_T) = 0$. For $\lambda = 1$ the recursion reads $G_t = r_t + \gamma G_{t+1}$, which is true.

The sum of TD errors. Subtract $V(s_t)$ from the recursion and add and subtract $\gamma\lambda V(s_{t+1})$:

$$G_t^\lambda - V(s_t) = \underbrace{r_t + \gamma V(s_{t+1}) - V(s_t)}_{\delta_t} + \gamma\lambda\big[G_{t+1}^\lambda - V(s_{t+1})\big].$$

Applying this to $t+1, t+2, \ldots$ and using $G_T^\lambda - V(s_T) = 0$ gives the sum. ■

Theorem 7.6 Offline equivalence of the forward and backward views

Consider one episode $s_0, a_0, r_0, \ldots, s_T$ and a value table V that is held fixed during the episode, with $V(s_T) = 0$ if s_T is terminal. Let z_t be the accumulating traces (7.12) and δ_t the TD errors computed with V. Then for every state s and all $\gamma, \lambda \in [0,1]$,

$$\sum_{t=0}^{T-1} \eta\,\delta_t\, z_t(s) = \sum_{t=0}^{T-1} \eta\,\big[G_t^\lambda - V(s_t)\big]\,\mathbb{1}\{s_t = s\}.$$

That is, offline TD(λ) and the offline λ-return algorithm change every state by the same total amount.

Proof. Insert $z_t(s) = \sum_{k=0}^{t} (\gamma\lambda)^{t-k}\mathbb{1}\{s_k = s\}$ and exchange the order of summation:

$$\sum_{t=0}^{T-1} \delta_t z_t(s) = \sum_{t=0}^{T-1}\sum_{k=0}^{t} (\gamma\lambda)^{t-k}\mathbb{1}\{s_k = s\}\,\delta_t = \sum_{k=0}^{T-1} \mathbb{1}\{s_k = s\}\sum_{t=k}^{T-1} (\gamma\lambda)^{t-k}\delta_t = \sum_{k=0}^{T-1} \mathbb{1}\{s_k = s\}\big[G_k^\lambda - V(s_k)\big],$$

by Lemma 7.5. Multiply by η. ■

Three remarks. The proof never used that s_T is terminal: if the episode is cut off by a time limit, the same statement holds with the truncated λ-return that bootstraps from $V(s_T)$, which

is exactly what $G_T^\lambda = V(s_T)$ gives. The theorem needs accumulating traces; with replacing traces the two sides differ whenever a state is visited twice, as in Example 7.5 (Exercise 7.10). And the theorem is the classical justification for TD(λ) (see Sutton and Barto 2018, ch. 12): the backward view is not a heuristic approximation of the λ-return, it *is* the λ-return, computed incrementally.

7.6.2 What changes online

Online TD(λ) changes V after every step, so the TD error δ_k is computed with the latest table V_k rather than with the table from the start of the episode. Identity (7.11) then no longer holds exactly. Every change of V during the episode is of size $O(\eta)$, so it changes the later TD errors by $O(\eta)$, and their contribution to the updates, which carries another factor η, by $O(\eta^2)$. For small step sizes, online TD(λ) therefore tracks the offline λ-return algorithm closely; for large ones it does not, and the divergence in the left panel of Figure 7.5 is the result.

What would an exact online forward view look like? At time h only the rewards up to r_{h-1} are known, so the best available target for an earlier time t is the *truncated λ-return*, which treats h as if it were the end of the episode and bootstraps there:

$$G_{t:h}^\lambda = (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1}\, G_{t:t+n} + \lambda^{h-t-1}\, G_{t:h}.$$

The *online λ-return algorithm* (Seijen and Sutton 2014) uses these targets for linear values $V_w(s) = w^\top x$ with feature vectors $x_t = \phi(s_t)$ (one-hot vectors for a table). At every time h it restarts from the weights w_0 that the episode began with and redoes all the updates $w \leftarrow w + \eta\,[G_{t:h}^\lambda - w^\top x_t]\,x_t$ for $t = 0, \dots, h-1$, where every n-step return $G_{t:t+n}$ bootstraps with the weights reached at horizon $t+n-1$. The result is the weight vector w_h. When the episode ends, the truncated returns are the full λ-returns, so this is the forward view, computed online. It costs $O(h)$ work at time h, which is impractical.

7.6.3 True online TD(λ)

Remarkably, the weights of the online λ-return algorithm can be computed exactly at the cost of TD(λ).

Algorithm 7.5 True online TD(λ) for linear values

Input: feature map ϕ with ϕ(terminal) $= 0$, a policy π, $\lambda \in [0,1]$, step size $\eta > 0$, discount γ.

1. Initialize w (for example $w = 0$).
2. For each episode:
   1. Observe s_0; $x \leftarrow \phi(s_0)$; $z \leftarrow 0$; $V_{\text{old}} \leftarrow 0$.
   2. For $t = 0, 1, 2, \dots$ until the episode ends:
      1. Take $a_t \sim \pi(\cdot \mid s_t)$; observe r_t and s_{t+1}; $x' \leftarrow \phi(s_{t+1})$.
      2. $V \leftarrow w^\top x$; $V' \leftarrow w^\top x'$; $\delta \leftarrow r_t + \gamma V' - V$.
      3. $z \leftarrow \gamma\lambda\, z + (1 - \eta\gamma\lambda\, z^\top x)\, x$.

4. $w \leftarrow w + \eta(\delta + V - V_{\text{old}})\, z - \eta(V - V_{\text{old}})\, x$.
5. $V_{\text{old}} \leftarrow V'$; $x \leftarrow x'$.

Output: w, with $V(s) \approx w^\top \phi(s)$.

Proposition 7.7 True online TD(λ) is the online λ-return algorithm

With linear values, the weights computed by Algorithm 7.5 after every step h of an episode equal the weights w_h of the online λ-return algorithm (Seijen and Sutton 2014; Seijen et al. 2016).

Proof. A sketch, in four steps. (1) When the horizon grows by one, every truncated λ-return changes by a single TD-like term: subtracting the definitions gives $G^\lambda_{t:h+1} - G^\lambda_{t:h} = (\gamma\lambda)^{h-t}\,\delta'_h$, with $\delta'_h = r_h + \gamma w_h^\top x_{h+1} - w_{h-1}^\top x_h$. (2) Writing $A_t = I - \eta x_t x_t^\top$, the redone updates unroll to $w_h = A_{h-1}\cdots A_0 w_0 + \eta \sum_{t<h} A_{h-1}\cdots A_{t+1}\, x_t\, G^\lambda_{t:h}$. (3) Subtracting this expression for h from the one for $h+1$, the changes of all the truncated returns combine into $\eta\,\delta'_h\,(z_h - x_h)$, where $z_h = \sum_{t\le h}(\gamma\lambda)^{h-t} A_h \cdots A_{t+1} x_t$; this sum satisfies $z_h = \gamma\lambda A_h z_{h-1} + x_h$, which is the trace update of step 3 of the algorithm. (4) Writing $\delta'_h = \delta_h + w_h^\top x_h - w_{h-1}^\top x_h$, with the ordinary TD error δ_h, and collecting terms gives $w_{h+1} = w_h + \eta(\delta_h + V - V_{\text{old}})\, z_h - \eta(V - V_{\text{old}})\, x_h$ with $V = w_h^\top x_h$ and $V_{\text{old}} = w_{h-1}^\top x_h$: step 4 of the algorithm. The full calculation is in Seijen et al. (2016), and Exercise 7.12 works through step (3). ■

For tabular (one-hot) features the trace of step 3 is the dutch trace of Section 7.5.1: the trace of the current state becomes $(1-\eta)\gamma\lambda z_{t-1}(s_t) + 1$. The extra terms with $V - V_{\text{old}}$ correct for the fact that the estimate of s_t has changed since it was last used as a bootstrap target. True online TD(λ) costs the same as TD(λ), tolerates much larger step sizes (Figure 7.5), and is the recommended version with linear function approximation (Seijen et al. 2016). The exact equivalence does not carry over to nonlinear function approximation. The tests of Lab 7.1 check the proposition numerically: the two algorithms agree to within rounding error at every step.

7.6.4 Convergence

Tabular TD(λ) converges with probability one to V^π for every $\lambda \in [0,1]$, provided every state is visited infinitely often and the step sizes satisfy the conditions of Section 6.9 (Dayan 1992; Jaakkola et al. 1994; Tsitsiklis 1994); the general stochastic-approximation theorem is in Appendix B. With linear function approximation and on-policy sampling in a continuing task, TD(λ) converges with probability one to the fixed point of a projected λ-version of the Bellman equation (Tsitsiklis and Van Roy 1997), and its error obeys

$$\left\| \Phi w_\lambda - V^\pi \right\|_d \le \frac{1-\gamma\lambda}{1-\gamma} \left\| \Pi V^\pi - V^\pi \right\|_d ,$$

where Φ is the matrix of features, $\|\cdot\|_d$ the norm weighted by the stationary distribution of the policy, and Π the projection onto the span of the features (Section 9.4). At $\lambda = 1$ the factor is one, the best that the features allow; at $\lambda = 0$ it is $1/(1-\gamma)$. This is one reason why values of λ close to one are popular with function approximation. (Chapter 9 gives a sharper bound for $\lambda = 0$.)

7.7 Control with eligibility traces ●●

7.7.1 SARSA(λ)

For on-policy control, traces live on state–action pairs, and the TD error is SARSA's.

Algorithm 7.6 SARSA(λ)

Input: $\lambda \in [0,1]$, step size $\eta \in (0,1]$, exploration rate $\varepsilon > 0$, discount γ.

1. Initialize $Q(s,a)$ arbitrarily, with $Q(\text{terminal},\cdot) = 0$.
2. For each episode:
   1. Observe s; choose a ε-greedily with respect to Q; set $z(x,y) \leftarrow 0$ for all pairs.
   2. Repeat until s is terminal:
      1. Take a; observe r and s'; choose a' ε-greedily with respect to Q in s'.
      2. $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$ (with $Q(s',a') = 0$ if s' is terminal).
      3. $z(s,a) \leftarrow z(s,a) + 1$ (accumulating) or $z(s,a) \leftarrow 1$ (replacing).
      4. For all pairs (x,y): $Q(x,y) \leftarrow Q(x,y) + \eta\,\delta\, z(x,y)$ and $z(x,y) \leftarrow \gamma\lambda\, z(x,y)$.
      5. $s \leftarrow s'$; $a \leftarrow a'$.

Output: Q, and the ε-greedy policy with respect to it.

Singh and Sutton (1996) also proposed clearing the traces of the other actions in the visited state when a replacing trace is set, so that only the action actually taken is credited. In the maze of Example 7.2, with $\lambda = 0.8$, $\eta = 0.3$ and the same 20 seeds, SARSA(0.8) needs 1,848 steps for the first 30 episodes with replacing traces and 1,894 with accumulating traces, against 2,727 for 4-step SARSA and 5,084 for one-step SARSA. At the end of the first episode the traces credit every pair on the long wandering path, with weights that fall by $\gamma\lambda = 0.76$ per step back from the goal.

7.7.2 Watkins's Q(λ)

For off-policy control the target policy is greedy and the behavior is exploratory. The λ-return of the greedy policy can follow the behavior's path only as long as the behavior acts greedily. At the first exploratory action the target must stop and bootstrap from $\max_a Q$, which is the tree-backup argument with a greedy π. *Watkins's Q(λ)* (Watkins 1989) does exactly this: Q-learning's TD errors, accumulating traces that fade by $\gamma\lambda$ after greedy actions, and traces cut to zero after an exploratory one.

Algorithm 7.7 Watkins's Q(λ)

Input: $\lambda \in [0,1]$, step size $\eta \in (0,1]$, exploration rate $\varepsilon > 0$, discount γ.

1. Initialize $Q(s, a)$ arbitrarily, with $Q(\text{terminal}, \cdot) = 0$.
2. For each episode:
   1. Observe s; choose a ε-greedily with respect to Q; set $z(x, y) \leftarrow 0$ for all pairs.
   2. Repeat until s is terminal:
      1. Take a; observe r and s'; choose a' ε-greedily with respect to Q in s'.
      2. $\delta \leftarrow r + \gamma \max_y Q(s', y) - Q(s, a)$ (with the maximum $= 0$ if s' is terminal).
      3. $z(s, a) \leftarrow z(s, a) + 1$; $Q(x, y) \leftarrow Q(x, y) + \eta\, \delta\, z(x, y)$ for all pairs.
      4. If a' is greedy, that is $Q(s', a') = \max_y Q(s', y)$: $z \leftarrow \gamma\lambda\, z$; else $z \leftarrow 0$.
      5. $s \leftarrow s'$; $a \leftarrow a'$.

Output: $Q \approx Q^*$ and the greedy policy.

Its weakness is that exploration keeps cutting the traces. With ε-greedy behavior over four actions, one of them greedy, and $\varepsilon = 0.1$, a trace survives each step with probability $1 - \varepsilon + \varepsilon/4 = 0.925$, so a stretch of greedy actions lasts about $1/0.075 \approx 13$ steps on average, and early in learning, when the greedy action keeps changing, much less. Whether the algorithm converges in general remained open for a long time; Munos et al. (2016) proved its convergence as a corollary of their analysis of Retrace (Section 7.8).

7.7.3 Peng's Q(λ)

Peng's Q(λ) (Peng and Williams 1996) never cuts traces. Every n-step component of its target follows the behavior's actual rewards and bootstraps with a maximum, so its forward-view target satisfies

$$G_t^{\lambda,\max} = r_t + \gamma\big[(1-\lambda)\max_a Q(s_{t+1}, a) + \lambda\, G_{t+1}^{\lambda,\max}\big], \tag{7.14}$$

the recursion (7.10) with a greedy bootstrap. It is neither on-policy nor exactly off-policy. With a fixed exploratory behavior it learns a mixture of the behavior's values and the optimal ones, and only if the behavior becomes greedy does the mixture tend to Q^*.

On the Trail with uniform behavior, $\gamma = 0.9$ and $\lambda = 0.5$, suppose $Q(\text{Middle}, \text{go}) = 3.6$ and $Q(\text{Goal}, \text{finish}) = 4$ are already learned (they equal Q^*, and Peng's fixed point agrees there). For the pair (Start, go), the continuation from Middle is $G_1 = 0.9\,(0.5 \cdot 4 + 0.5 \cdot 4) = 3.6$ if the behavior goes and $G_1 = 2$ if it exits, so $\mathbb{E}[G_1] = 2.8$ and the fixed point is

$$Q(\text{Start}, \text{go}) = 0.9\big(0.5 \cdot 3.6 + 0.5 \cdot 2.8\big) = 2.88,$$

between the behavior's value 2.52 and the optimal value 3.24. In practice Peng's Q(λ) often learns faster than Watkins's because it never throws away the sampled return. Kozuno et al. (2021) proved that it converges to an optimal policy when the behavior policy slowly tracks the greedy one, and found it competitive in deep reinforcement-learning experiments. Its backward-view implementation is more involved than Watkins's (Peng and Williams 1996); when whole episodes are stored, the target (7.14) is computed in one backward pass.

7.8 Off-policy traces with guarantees: Retrace and V-trace ●●●

Importance sampling, tree backup, Watkins's and Peng's Q(λ) all answer one question: how much of the sampled return should an off-policy target use? This section puts them into a single form, states the guarantee that singles out Retrace, and analyzes V-trace, the state-value counterpart used in large distributed agents.

7.8.1 One operator, four choices of trace

Following Munos et al. (2016), consider targets of the form $Q(s_t,a_t)+\sum_{k\ge t}\gamma^{k-t}\big(\prod_{i=t+1}^{k}c_i\big)\,\delta_k^{\pi}$, where $\delta_k^{\pi}=r_k+\gamma\bar V(s_{k+1})-Q(s_k,a_k)$ is the Expected SARSA TD error under the target policy and the *trace coefficients* $c_i=c(s_i,a_i)\ge 0$ decide how far the sampled path is followed. Their expectation under the behavior policy defines an operator on action-value functions:

$$(\mathcal{R}Q)(s,a)=Q(s,a)+\mathbb{E}_b\Big[\sum_{t\ge 0}\gamma^t\Big(\prod_{i=1}^{t}c_i\Big)\big(r_t+\gamma\bar V(s_{t+1})-Q(s_t,a_t)\big)\;\Big|\;s_0=s,\;a_0=a\Big]. \tag{7.15}$$

The sampled targets are computed backward by $G_t=r_t+\gamma\big[\bar V(s_{t+1})+c_{t+1}\,(G_{t+1}-Q(s_{t+1},a_{t+1}))\big]$. Table 7.2 lists four choices.

Table 7.2. Four ways to set the traces of the off-policy operator (7.15), after Munos et al. (2016). "Safe" means that the operator contracts toward Q^{π} for every behavior policy.

Method	trace c_i	safe for any b?	trace when $\pi=b$	variance
importance sampling	$\lambda\rho_i$	yes	λ	can be infinite
tree backup, TB(λ)	$\lambda\,\pi(a_i\mid s_i)$	yes	$\lambda\pi(a_i\mid s_i)<\lambda$	small
$Q^{\pi}(\lambda)$	λ	only if $\pi\approx b$	λ	small
Retrace(λ)	$\lambda\min(1,\rho_i)$	yes	λ	small

Three properties are at stake. A choice is *safe* if the expected operator converges to Q^{π} whatever the behavior policy. It is *efficient* if it uses long stretches of the sampled return when the data are nearly on-policy, so that the operator contracts quickly. And it has *low variance* if the products of traces stay bounded. Importance sampling is safe and efficient but its products can explode. Tree backup is safe with small variance, but it cuts traces even on-policy. $Q^{\pi}(\lambda)$, which uses the return without any correction, is efficient and steady, but only safe when the policies are close (Harutyunyan et al. 2016).

7.8.2 Retrace(λ)

Retrace(λ) truncates the ratios at one: $c_i=\lambda\min(1,\rho_i)$. It has all three properties at once.

Theorem 7.8 Safe off-policy returns (Munos et al., 2016)

Suppose $b(a \mid s) > 0$ wherever $\pi(a \mid s) > 0$, and the trace coefficients satisfy $0 \le c(s,a) \le \pi(a \mid s)/b(a \mid s)$ for all s, a. Then Q^π is the unique fixed point of $\mathcal{R}$, and for every Q,

$$\|\mathcal{R}Q - Q^\pi\|_\infty \le \kappa \|Q - Q^\pi\|_\infty, \qquad \kappa = \max_{s,a}\Big[1 - (1-\gamma)\,\mathbb{E}_b\Big[\sum_{t\ge 0}\gamma^t \prod_{i=1}^{t} c_i \;\Big|\; s_0 = s, a_0 = a\Big]\Big] \le \gamma.$$

Proof. Define two operators on functions of state–action pairs: $(P^\pi f)(s,a) = \sum_{s'} P(s' \mid s,a)\sum_{a'}\pi(a' \mid s')f(s',a')$, and P^{cb}, the same with $\pi(a' \mid s')$ replaced by $b(a' \mid s')\,c(s',a')$. By the Markov property, $\mathbb{E}_b[\prod_{i=1}^{t} c_i\, f(s_t,a_t) \mid s_0 = s, a_0 = a] = ((P^{cb})^t f)(s,a)$, and the expected TD error at time t given (s_t,a_t) is $(\mathcal{T}^\pi Q - Q)(s_t,a_t)$, with the action-value Bellman operator $\mathcal{T}^\pi Q = r + \gamma P^\pi Q$. So $\mathcal{R}Q = Q + \sum_{t\ge 0}\gamma^t (P^{cb})^t(\mathcal{T}^\pi Q - Q)$, and $\mathcal{R}Q^\pi = Q^\pi$ because $\mathcal{T}^\pi Q^\pi = Q^\pi$.

Let $\Delta = Q - Q^\pi$. Then $\mathcal{T}^\pi Q - Q = \gamma P^\pi \Delta - \Delta$, and

$$\mathcal{R}Q - Q^\pi = \Delta + \sum_{t\ge 0}\gamma^t (P^{cb})^t(\gamma P^\pi \Delta - \Delta) = \sum_{t\ge 0}\gamma^{t+1}(P^{cb})^t\big(P^\pi - P^{cb}\big)\Delta,$$

because the terms $-\sum_{t\ge 0}\gamma^t (P^{cb})^t\Delta$ and $\Delta + \sum_{t\ge 0}\gamma^{t+1}(P^{cb})^t P^\pi\Delta$ regroup after shifting the index of the first sum by one. The operator $P^\pi - P^{cb}$ has the weights $P(s' \mid s,a)\,[\pi(a' \mid s') - b(a' \mid s')\,c(s',a')]$, which are nonnegative by the assumption on c. So $\mathcal{R}Q - Q^\pi = M\Delta$ with a nonnegative operator M, and $|(\mathcal{R}Q - Q^\pi)(s,a)| \le (M1)(s,a)\,\|\Delta\|_\infty$, where 1 is the constant function 1. With $p_t = (P^{cb})^t 1$, which is the expected product of the first t traces, $P^\pi 1 = 1$ gives

$$M1 = \sum_{t\ge 0}\gamma^{t+1}(p_t - p_{t+1}) = \gamma - (1-\gamma)\sum_{t\ge 1}\gamma^t p_t = 1 - (1-\gamma)\sum_{t\ge 0}\gamma^t p_t,$$

using $p_0 = 1$. This is the bracket in the definition of κ, and it is at most γ because the p_t are nonnegative. The series converge because $b\,c \le \pi$ makes $p_t \le 1$. Uniqueness of the fixed point follows from the contraction. ■

Retrace satisfies the condition of the theorem for every behavior policy, so it is safe. Near on-policy, $\rho_i \ge 1$ for the actions the target prefers, so their traces equal λ and long stretches of return are used: it is efficient. And every $c_i \le \lambda \le 1$, so the products never grow: its variance is small. Truncating at one is the largest choice that keeps the condition $c \le \rho$ for every behavior, while the products stay bounded. The theorem also covers the other two safe rows of Table 7.2, and its constant shows what the choice of trace buys: larger traces give a smaller κ, that is, faster contraction. Full importance sampling, $c = \rho$, gives $\kappa = 0$: one application of the expected operator returns Q^π exactly, which is the statement that the importance-weighted return is unbiased. Cutting all traces, $c = 0$, gives the one-step backup $\mathcal{T}^\pi$ and $\kappa = \gamma$. With a greedy target policy, Retrace's trace is λ after greedy actions and 0 after the others: Watkins's Q(λ). Munos et al. (2016) extended the analysis to control, with target policies that become greedy with respect to the current estimates, proved convergence to Q^* without requiring the behavior to become greedy, and obtained the convergence of Watkins's Q(λ) as a corollary.

Figure 7.6 makes these properties visible on a small problem, computed exactly with the matrix form of the operator (Exercise 7.16). At $\lambda = 1$ the rates of the expected operators are 0 for

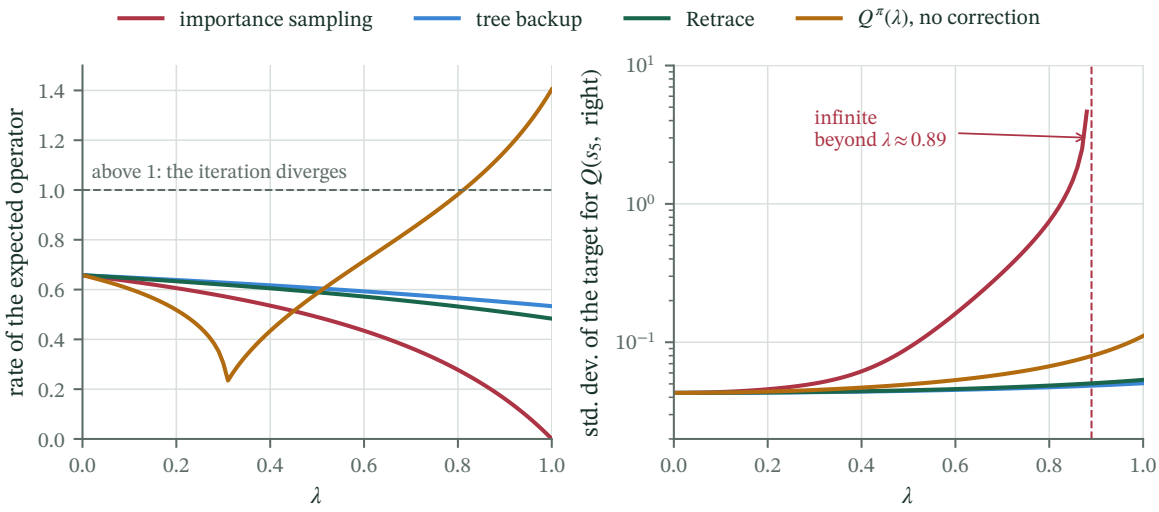


Figure 7.6. Speed and variance of off-policy operators on a corridor of 9 states ($\gamma = 0.9$; each move slips the other way with probability 0.1; the target policy moves right with probability 0.9, the behavior with probability 0.3). Left: the rate of the exact expected operator, the factor by which it eventually shrinks the error per application; above 1 the iteration diverges. Right: standard deviation of the sampled target for one pair at the fixed point (log scale). Importance sampling contracts fastest but its variance becomes infinite; $Q^\pi(\lambda)$ diverges for large λ; Retrace and tree backup are stable, and Retrace is the faster of the two.

importance sampling, 0.48 for Retrace and 0.53 for tree backup, while $Q^\pi(\lambda)$ has rate 1.40 and diverges; its rate crosses one near $\lambda = 0.81$. The standard deviations of the sampled targets tell the other half of the story: about 0.05 for Retrace and tree backup, 0.11 for $Q^\pi(\lambda)$ at $\lambda = 1$, and infinite for importance sampling from $\lambda \approx 0.89$ on, because products of ratios 3 and 1/7 grow faster than discounting and termination shrink them.

7.8.3 V-trace

Large distributed agents such as IMPALA (Espeholt et al. 2018) learn a state-value function from trajectories generated by many actors whose policies lag behind the learner's. Their *V-trace* target truncates two kinds of ratio, $\rho_k = \min(\bar\rho, \pi(a_k \mid s_k)/b(a_k \mid s_k))$ and $c_k = \min(\bar c, \pi(a_k \mid s_k)/b(a_k \mid s_k))$ with levels $\bar\rho \geq \bar c$:

$$v_t = V(s_t) + \sum_{k=t}^{t+n-1} \gamma^{k-t} \Big(\prod_{i=t}^{k-1} c_i\Big) \rho_k \big(r_k + \gamma V(s_{k+1}) - V(s_k)\big), \tag{7.16}$$

computed backward with $v_t = V(s_t) + \rho_t \delta_t + \gamma c_t\,(v_{t+1} - V(s_{t+1}))$. Unlike Retrace, the action a_t itself needs a correction here, because $V(s_t)$ averages over actions. On-policy, with $\bar c \geq 1$, every truncated ratio is one and v_t is the n-step return.

Proposition 7.9 What V-trace learns

Let $\pi_{\bar\rho}(a \mid s) = \min(\bar\rho\, b(a \mid s), \pi(a \mid s))/\sum_{a'} \min(\bar\rho\, b(a' \mid s), \pi(a' \mid s))$. A table V is a fixed point of the expected V-trace update if and only if $V = V^{\pi_{\bar\rho}}$, the value function of $\pi_{\bar\rho}$.

Proof. As in the proof of Theorem 7.8, the expected V-trace target is $V + (I - \gamma M_c)^{-1} e$, where $M_c(s, s') = \sum_a b(a \mid s)\, c(s, a)\, P(s' \mid s, a)$ and $e(s) = \sum_a b(a \mid s)\, \rho(s, a)\, [Q_V(s, a) - V(s)]$ with $Q_V(s, a) = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V(s')$. Because $b\, c \le \pi$, the rows of M_c sum to at most one, so $I - \gamma M_c$ is invertible and the target equals V exactly when $e = 0$. Now $b(a \mid s)\, \rho(s, a) = \min(\bar\rho\, b(a \mid s), \pi(a \mid s))$, so dividing $e(s) = 0$ by the normalizing sum gives $V(s) = \sum_a \pi_{\bar\rho}(a \mid s)\, Q_V(s, a)$: the Bellman equation of $\pi_{\bar\rho}$, whose unique solution is $V^{\pi_{\bar\rho}}$. ■

So the level $\bar\rho$ decides *which* policy is evaluated, and $\bar c$ only decides how fast. If $\bar\rho \ge \max \pi/b$, nothing is truncated inside the target and $\pi_{\bar\rho} = \pi$; with smaller $\bar\rho$ the evaluated policy moves toward the behavior. A deterministic target policy is never distorted, since $\min(\bar\rho b, \pi)$ is then nonzero only for the action π takes. Espeholt et al. (2018) also prove that, with $\bar\rho \ge \bar c$ and truncated ratios whose expectation is bounded away from zero, the expected V-trace operator is a sup-norm contraction, so it converges to this fixed point. Chapter 13 shows how IMPALA uses the V-trace targets in an actor-critic.

7.8.4 Speed, variance and bias

The off-policy multi-step methods trade three quantities: the contraction rate of the expected operator, the variance of the sampled targets, and the bias of the fixed point. Importance sampling has the fastest contraction and an unbiased fixed point but unbounded variance. Retrace and tree backup keep the fixed point Q^π and control the variance by cutting traces, which slows the contraction. V-trace truncates inside the target as well, which lowers the variance further but moves the fixed point to $V^{\pi_{\bar\rho}}$. Uncorrected returns, $Q^\pi(\lambda)$ and the n-step returns of many deep agents, are the most aggressive: low variance, fast propagation, a fixed point biased toward the behavior's values, and, far from on-policy, possible divergence. Rowland et al. (2020) make this three-way trade-off explicit and propose adapting the traces to sit at a chosen point of it.

RESEARCH NOTE

Open questions. How should λ (or n) be adapted online, per state or per time step, rather than tuned by hand? Which off-policy corrections remain safe with function approximation, where even on-policy bootstrapping interacts with the deadly triad (Section 9.8)? Why do non-conservative methods such as Peng's Q(λ) and uncorrected n-step returns, which lack Retrace's guarantees, often perform as well or better in deep reinforcement learning (Kozuno et al. 2021; Hessel et al. 2018)? And how should multi-step targets be combined with large replay buffers, whose data come from many old policies (Fedus et al. 2020)?

7.9 Multi-step targets in deep reinforcement learning

Multi-step returns are among the most widely used ideas of this chapter. Here is where they appear in the deep methods of later chapters, and what to watch out for when you use them.

Value-based agents. Deep Q-networks (Algorithm 10.1) are often trained with an n-step target, $\sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_a Q_{\bar{w}}(s_{t+n}, a)$, where $Q_{\bar{w}}$ is the target network. Rainbow uses $n = 3$, and its ablations found multi-step learning to be one of its two most important components, together with prioritized replay (Hessel et al. 2018) (Section 10.7). These targets are not corrected for off-policyness: the replayed rewards were earned by older, more exploratory policies, so the target mixes the behavior's values with the greedy ones, much like Peng's Q(λ). For small n the bias is usually worth the faster propagation. Fedus et al. (2020) found that n-step returns are what lets such agents benefit from larger replay buffers.

Actor-critic methods. A2C and A3C (Algorithm 11.3) estimate advantages with n-step returns over short segments of experience (Mnih et al. 2016). *Generalized advantage estimation* (GAE, Section 11.7) uses the λ-return instead: its advantage estimate $\hat{A}_t = \sum_{l \geq 0} (\gamma\lambda)^l \delta_{t+l}$ ((11.5)) is, by identity (7.11), exactly $G_t^\lambda - V(s_t)$, the λ-return error of the critic (Schulman et al. 2016). PPO computes it for every collected batch (Algorithm 12.2); the original PPO experiments used $\lambda = 0.95$ with $\gamma = 0.99$ (Schulman, Wolski, et al. 2017). IMPALA's V-trace (Section 7.8.3) is the off-policy version of the same idea.

Choosing n and λ. No value is right for every problem, but some patterns hold. Longer lookahead helps most when rewards are sparse and delayed, when value estimates are poor (early in learning), and when function approximation makes bootstrapping inaccurate. Shorter lookahead helps when returns are noisy, when the estimates are already good, and when the data are far off-policy. Tune n or λ together with the step size, as Figure 7.2 showed, and remember that λ corresponds roughly to $n = 1/(1-\lambda)$.

PITFALL

Episode boundaries in multi-step targets. In a replay buffer or a batch of rollouts, consecutive transitions may belong to different episodes. An n-step sum must stop at the end of its episode: after a true termination there is no bootstrap, after a time-limit truncation there is one, from the last state of the episode (Section 6.8). Summing rewards across a reset, or bootstrapping from the first state of the next episode, silently corrupts the targets. The same applies to traces: reset z to zero at the start of every episode.

PITFALL

The ratios of off-policy SARSA. In (7.6) the product of ratios starts at $t+1$, not at t, and ends at $t+n$, not at $t+n-1$. Including ρ_t wrongly reweights the action being evaluated; leaving out ρ_{t+n} makes the bootstrap term follow the behavior's choice of last action instead of the target's.

7.10 Summary

- The n-step return uses n real rewards and then the estimated value of the state reached; $n = 1$ gives the TD target and large n the Monte Carlo return. Its bias shrinks like γ^n (the error-reduction property), while its variance grows with n.
- n-step TD and n-step SARSA delay each update by $n-1$ steps and carry information back n states per episode. On the random walk with 19 states, $n = 4$ beat both one-step TD

and Monte Carlo; the best n depends on the problem, the step size and the amount of experience.

- Off-policy n-step learning needs a correction: importance-sampling ratios (unbiased, but high variance) or tree backup (no ratios, but it stops following the sampled path where the target policy would not). Q(σ) mixes the two.
- The λ-return averages all n-step returns with weights $(1-\lambda)\lambda^{n-1}$; $\lambda = 0$ is TD(0) and $\lambda = 1$ is Monte Carlo. Its error is a discounted sum of TD errors, $G_t^\lambda - V(s_t) = \sum_k (\gamma\lambda)^{k-t}\delta_k$.
- Eligibility traces turn this sum into an online algorithm, TD(λ): each TD error is shared among recently visited states in proportion to their fading traces. Accumulating, replacing and dutch traces differ in how they treat repeated visits.
- (●●) Offline, TD(λ) with accumulating traces equals the λ-return algorithm exactly; true online TD(λ) reproduces the online λ-return algorithm exactly with linear function approximation and is robust to large step sizes. SARSA(λ), Watkins's Q(λ) and Peng's Q(λ) bring traces to control.
- (●●●) A single operator with trace coefficients c covers importance sampling, tree backup, $Q^\pi(\lambda)$ and Retrace. Retrace, $c = \lambda \min(1, \rho)$, contracts toward Q^π for any behavior with bounded variance; V-trace evaluates a policy between the target and the behavior.
- Multi-step targets are everywhere in deep reinforcement learning: n-step DQN and Rainbow, GAE in PPO, V-trace in IMPALA.

The next chapter adds the last ingredient of the tabular toolbox: learning a model of the environment and using it to plan, so that each real experience can be replayed and reused many times.

7.11 Further Reading

The standard textbook treatment is Sutton and Barto (2018), Chapter 7 for n-step methods and Chapter 12 for eligibility traces, including traces with function approximation, true online methods and off-policy traces; its bibliographical remarks trace the history in detail. Szepesvári (2010) gives a compact, mathematically careful account of TD(λ).

The original papers are readable. Sutton (1988) introduced TD(λ) and analyzed it as a method for learning to predict. Watkins (1989) formulated the forward view with n-step returns and introduced Q-learning together with Q(λ), and Peng and Williams (1996) gave the alternative Q(λ) that bears Peng's name. Singh and Sutton (1996) introduced replacing traces and related accumulating and replacing traces to every-visit and first-visit Monte Carlo. Precup et al. (2000) introduced tree backup and per-decision importance sampling for off-policy evaluation, and De Asis et al. (2018) unified them as Q(σ). Seijen and Sutton (2014) introduced the online λ-return algorithm and true online TD(λ); Seijen et al. (2016) is the full journal version, with the derivation, extensions to control and extensive experiments. For convergence, Dayan (1992), Jaakkola et al. (1994) and Tsitsiklis (1994) treat the tabular case and Tsitsiklis and Van Roy (1997) the linear one.

(●●●) Munos et al. (2016) is the key paper on off-policy traces: it introduces Retrace, proves Theorem 7.8 in a more general form with history-dependent traces, and handles control. Haru-

tyunyan et al. (2016) analyze $Q^\pi(\lambda)$ and when it is safe. Espeholt et al. (2018) introduce V-trace and the IMPALA architecture. Rowland et al. (2020) make the trade-off between contraction, variance and fixed-point bias explicit, and Kozuno et al. (2021) revisit Peng's Q(λ) for deep reinforcement learning. On the theory of choosing n and λ, Kearns and Singh (2000) prove bias–variance error bounds for phased versions of n-step TD and TD(λ) and derive schedules for n and λ that their bounds predict will beat any fixed value. Schulman et al. (2016) derive generalized advantage estimation from λ-returns, and Hessel et al. (2018) and Fedus et al. (2020) document the role of n-step returns in deep Q-learning.

7.12 Exercises

Exercise 7.1 • n-step returns by hand

On the Trail with $\gamma = 0.9$, the uniform policy produces the episode Start, go, Middle, exit (rewards 0 and 2). (a) Compute $G_{0:1}$, $G_{0:2}$ and $G_{0:3}$ for $s_0 = \text{Start}$, first with $V = 0$ and then with the true values $V = V^\pi = (1.76, 2.8, 4)$. (b) With $V = 0$, compute the expected value of $G_{0:1}$, $G_{0:2}$ and $G_{0:3}$ under the uniform policy and compare with $V^\pi(\text{Start}) = 1.76$. (c) With $V = V^\pi$, compute the variance of $G_{0:1}$ and of the Monte Carlo return G_0. What do (b) and (c) say about bias and variance?

Exercise 7.2 • The weights of the λ-return

(a) Show that the weights in (7.9) add up to one. (b) In a continuing task, show that the weights $(1-\lambda)\lambda^{n-1}$, $n = 1, 2, \ldots$, have mean $1/(1-\lambda)$. (c) For $\lambda = 0.9$, what fraction of the weight goes to n-step returns with $n \le 10$? (d) With $\gamma = 0.9$ and $\lambda = 0.9$, after how many steps has an eligibility trace fallen to half its value?

Exercise 7.3 • The two ends of TD(λ)

(a) Show that TD(λ) with $\lambda = 0$ is exactly TD(0), online and offline. (b) Show that offline TD(1) with accumulating traces and $\gamma = 1$ makes the every-visit Monte Carlo update with a constant step size. (c) TD(1) is sometimes described as "Monte Carlo that works in continuing tasks". Explain what that means and why the Monte Carlo method of Chapter 5 cannot do it.

Exercise 7.4 • Traces in the random walk

In the random walk with five states A, … , E (reward +1 only for leaving on the right, $\gamma = 1$), all estimates are 0.5. An episode visits C, D, C, B, A and then leaves on the left. With $\lambda = 0.5$ and $\eta = 0.1$: (a) compute the TD errors; (b) compute the accumulating and the replacing traces of every state at the last step; (c) compute the offline TD(λ) changes of all five estimates with each kind of trace; (d) check the accumulating result against the λ-returns.

Exercise 7.5 • Which ratios?

On the Trail, the target policy always goes and the behavior is uniform; finish is the only action in Goal. For each of the episodes (a) go, go, finish, (b) go, exit and (c) exit, give the ratio that 2-step off-policy TD uses to update $V(\text{Start})$ and the ratio that 2-step off-policy SARSA uses to update $Q(s_0, a_0)$. Which updates are skipped? In (c), explain why SARSA still learns something useful about $Q(\text{Start}, \text{exit})$.

Exercise 7.6 • A tree-backup target

On the Trail with $\gamma = 0.9$, let the target policy be $p_0 = 0.8$, $p_1 = 0.5$, and let $Q(\text{Middle}, \text{go}) = 3.6$, $Q(\text{Middle}, \text{exit}) = 2$ and $Q(\text{Goal}, \text{finish}) = 4$. (a) Compute the 2-step tree-backup target for (Start, go) on the episodes go, go, finish and go, exit. (b) Compute the 2-step importance-sampling target and weight for the same episodes, if the behavior is uniform. (c) Compare both with $Q^\pi(\text{Start}, \text{go})$ and explain what you see.

Exercise 7.7 • Find the bug

Each description is of a buggy implementation. Say what goes wrong and how to fix it. (a) A replay buffer stores transitions in order and forms 3-step targets from any three consecutive entries. (b) A TD(λ) agent creates its trace vector once, before the first episode. (c) An agent computes λ-returns with $G_T^\lambda = 0$ for every episode, including episodes cut off by a time limit of 500 steps. (d) Off-policy 4-step SARSA weights its update by $\rho_{t:t+3}$.

Exercise 7.8 • Does the best n depend on the budget?

Extend Task 2 of Lab 7.1. Run 100 episodes instead of 10 (use step sizes down to 0.01) and record two measures for $n = 1, 2, 4, 8, 16, 32$: the RMS error averaged over the 100 episodes, and the error after the 100th episode. Which n is best for each measure? Explain the difference.

Exercise 7.9 •• The error-reduction property

(a) Prove Proposition 7.2 in detail, including the induction step $\mathbb{E}_\pi[G_{t:t+n} \mid s_t = s] = ((\mathcal{T}^\pi)^n V)(s)$. (b) In an episodic task, show the sharper bound $|\mathbb{E}_\pi[G_{t:t+n} \mid s_t = s] - V^\pi(s)| \le \gamma^n \Pr_\pi(T > t+n \mid s_t = s) \max_{s'} |V(s') - V^\pi(s')|$. (c) For the Trail with the uniform policy and $V = 0$, compute both bounds for $n = 1$ and $n = 2$ at Start and compare them with the actual errors.

Exercise 7.10 •• Replacing traces and first visits

(a) Use Example 7.5 to show that offline TD(λ) with replacing traces does not equal the offline λ-return algorithm. (b) Show that with $\lambda = 1$ and $\gamma = 1$, offline TD(1) with replacing traces changes each visited state s by $\eta\,[G_{t_1} - V(s)]$, where t_1 is the time of the first visit: the first-visit Monte Carlo update. (c) What does the accumulating version give, and which estimator of Chapter 5 is it?

Exercise 7.11 ●● λ-returns, TD errors and GAE

(a) Prove identity (7.11) directly from the recursion (7.10). (b) An episode is cut off by a time limit at T, and s_T is not terminal. Show that the identity still holds for the truncated λ-return with $G_T^\lambda = V(s_T)$, and explain what goes wrong if $G_T^\lambda = 0$ is used instead. (c) For the episode Start, go, Middle, exit with $V = (1.76, 2.8, 4)$, $\gamma = 0.9$ and $\lambda = 0.5$, compute the GAE advantage $\hat{A}_0 = \delta_0 + \gamma\lambda\,\delta_1$ and check that $G_0^\lambda = V(\text{Start}) + \hat{A}_0$ by computing G_0^λ from its definition.

Exercise 7.12 ●● Where dutch traces come from

(a) Let $A_t = I - \eta x_t x_t^\top$ and $z_h = \sum_{t=0}^{h} (\gamma\lambda)^{h-t} A_h \cdots A_{t+1}\, x_t$ (the empty product for $t = h$ is I). Show that $z_h = \gamma\lambda\, z_{h-1} + (1 - \eta\gamma\lambda\, z_{h-1}^\top x_h)\, x_h$. (b) For one-hot features, show that the trace of the current state satisfies $z_h(s_h) = (1-\eta)\,\gamma\lambda\, z_{h-1}(s_h) + 1$, while the other traces fade by $\gamma\lambda$. (c) What is the dutch trace for $\eta = 1$? For $\eta \to 0$?

Exercise 7.13 ●● Importance-sampled n-step targets are unbiased

(a) For a fixed table V, prove that $\mathbb{E}_b[\rho_{t:t+n-1}\, G_{t:t+n} \mid s_t = s] = \mathbb{E}_\pi[G_{t:t+n} \mid s_t = s]$, assuming $b(a \mid s) > 0$ wherever $\pi(a \mid s) > 0$. (b) Prove the corresponding statement for off-policy n-step SARSA with the ratio $\rho_{t+1:t+n}$, conditioning on $s_t = s$ and $a_t = a$. (c) Explain why the update (7.5) multiplies the whole error $G_{t:t+n} - V(s_t)$ by the ratio, and not only the return.

Exercise 7.14 ●● Cut traces and mixed targets

(a) In Watkins's Q(λ) with ε-greedy behavior over $|\mathcal{A}|$ actions with a unique greedy action, find the probability that a trace survives k steps without being cut, and the expected length of an uncut stretch, for $|\mathcal{A}| = 4$ and $\varepsilon = 0.1$. (b) Show that the expected Peng's Q(λ) target defines the operator $Q \mapsto (1-\lambda)\sum_{n\geq 1} \lambda^{n-1} (\mathcal{T}^b)^{n-1} \mathcal{T}^* Q$, where $\mathcal{T}^b$ and $\mathcal{T}^*$ are the Bellman operators of the behavior and the optimality operator on action values, and that it is a contraction. (c) On the Trail with uniform behavior and $\gamma = 0.9$, compute Peng's fixed point $Q(\text{Start}, \text{go})$ for $\lambda = 0.9$, and its limits as $\lambda \to 0$ and $\lambda \to 1$.

Exercise 7.15 ●● Traces for control in the maze

In the maze of Example 7.2 ($\gamma = 0.95$, $\varepsilon = 0.1$), compare one-step SARSA, 4-step SARSA, SARSA(0.8) with replacing and with accumulating traces, and Watkins's Q(0.8) for $\eta \in \{0.1, 0.3, 0.5\}$, using the total number of steps in the first 30 episodes, averaged over 20 seeds. Which method is best, how sensitive is each to η, and why do the traced methods do so much better than 4-step SARSA after the first episode?

Exercise 7.16 ●●● The Retrace operator in matrix form

(a) For a finite MDP, write $\mathcal{R}$ of (7.15) as $\mathcal{R}Q = Q + (I - \gamma P^{cb})^{-1}(\mathcal{T}^{\pi}Q - Q)$ and verify numerically, on a random MDP, that Q^{π} is its fixed point and that $\|\mathcal{R}Q - Q^{\pi}\|_{\infty} \le \kappa\|Q - Q^{\pi}\|_{\infty}$ for random Q and Retrace traces. (b) Show that $\kappa = 0$ for $c = \rho$ and $\kappa = \gamma$ for $c = 0$, and explain both results. (c) Give a two-action example in which $c = \lambda$ violates the condition of Theorem 7.8, and explain which step of the proof fails. (d) Reproduce the left panel of Figure 7.6 with `rlcourse.nstep.operator_rate`.

Exercise 7.17 ●●● The bias of V-trace

A single state has two actions; action 1 earns 1 and action 2 earns 0, and the episode ends after one step. The target policy is $\pi = (0.9, 0.1)$ and the behavior $b = (0.5, 0.5)$. (a) Compute $\pi_{\bar{\rho}}$ and the V-trace fixed point for $\bar{\rho} = 1$ and for $\bar{\rho} = 2$, and compare with V^{π}. (b) For which $\bar{\rho}$ is V-trace unbiased in general? (c) Explain why $\bar{c}$ does not affect the fixed point, and what it does affect. (d) Why might a practitioner still choose $\bar{\rho} = 1$?

Exercise 7.18 ●●● Conservative or not?

Off-policy multi-step methods can be *conservative*, cutting traces when the behavior departs from the target (tree backup, Retrace, Watkins's Q(λ)), or *non-conservative*, following the behavior's return without correction (Peng's Q(λ), uncorrected n-step returns in Rainbow). Using Munos et al. (2016), Harutyunyan et al. (2016), Kozuno et al. (2021), Hessel et al. (2018) and Fedus et al. (2020), summarize what is proved and what is observed about each kind, explain why non-conservative methods can work well in practice despite their bias, and propose an experiment that would separate the effect of bias from the effect of faster propagation.

CHAPTER 8

Planning and Learning with Models

WHAT YOU WILL LEARN

- Distinguish distribution models from sample models, and build a certainty-equivalence model of a small problem from data.
- Explain how Dyna integrates acting, learning and planning, and implement Dyna-Q and Dyna-Q+.
- Use prioritized sweeping to focus planning on the updates that matter, and explain why it beats uniform planning.
- Compare expected and sample updates in cost and accuracy, and explain trajectory sampling and real-time dynamic programming (RTDP).
- Plan at decision time with rollout algorithms and Monte Carlo tree search, and compute UCT selections by hand.
- State what sparse sampling and UCT guarantee, and what planning in an estimated model costs (●●).
- Explain why tabular planning does not scale directly to deep models, and what modern methods do instead (●●●).

Every method of Chapters 5–7 learns from real experience only. Each transition is used for a handful of updates and then discarded, so the agent needs a great deal of experience: in the maze of Chapter 7, one-step SARSA needed thousands of steps before it walked straight to the goal. Dynamic programming (Chapter 4) is the opposite extreme. It needs no experience at all, but it needs a complete and correct description of the environment, which an agent rarely has.

This chapter combines the two. An agent can *learn a model* of its environment from experience and then *plan* with it: generate simulated experience from the model and learn from that too. Each real step then improves the model, and each improvement of the model can be replayed through many planning updates. The idea is old, and the architectures in this chapter, Dyna, prioritized sweeping and real-time dynamic programming, date from around 1990, but the same idea drives modern successes. AlphaGo and its successors plan with tree search; MuZero plans with a learned model; Dreamer learns its behavior almost entirely inside a learned world model (Chapter 15).

Two questions organize the chapter. *How* should planning updates be made: from full distributions or from samples, all at once or one state at a time? And *where* should the planning effort go: uniformly over all states, backward from surprising changes, along the trajectories the agent actually follows, or into a search tree rooted at the current state? A useful slogan is that planning is learning from simulated experience: almost every planning method below is one of the value

updates of Chapters 4–7, applied to experience that comes from a model.

Section 8.1 defines models and shows how to estimate one from data. Section 8.2 presents Dyna, which interleaves learning and planning, and what goes wrong when the model is out of date. Section 8.3 focuses planning with prioritized sweeping. Section 8.4 compares expected and sample updates, introduces trajectory sampling, and presents real-time dynamic programming. Section 8.5 turns to planning at decision time: heuristic search, rollout algorithms and Monte Carlo tree search. Section 8.6 (●●) states what can be proved about planning with sampled models, and Section 8.7 (●●●) explains why tabular Dyna does not scale directly to deep models and points to the methods that do. On a first reading, read Section 8.1 to 8.5.

8.1 Models and planning

A *model* of the environment is anything an agent can use to predict how the environment will respond to its actions: given a state and an action, a model predicts the next state and the reward. Models come in two kinds.

Definition 8.1 Distribution and sample models

A *distribution model* gives the probabilities of all possible outcomes: the transition probabilities $P(s' \mid s, a)$ and the expected rewards $r(s, a)$. A *sample model* produces one possible outcome: given (s, a) it returns a next state and a reward drawn with the correct probabilities.

Dynamic programming needs a distribution model, because its updates are expectations over all next states. A sample model is often much easier to obtain. A simulator of a robot or of a game is a sample model, and so is a physics engine. Writing down the probability of every outcome of rolling twelve dice is tedious; simulating the roll is easy. A distribution model can always produce samples; the reverse requires estimating probabilities.

Planning, in this chapter, means any computation that takes a model as input and produces or improves a policy (or the value function that defines one). The planning methods here all search the space of states and work through value functions: they compute values by updates, or backups, toward simulated targets, and act greedily with respect to them. Seen this way, dynamic programming is planning with a distribution model, and Q-learning applied to transitions drawn from a sample model is planning too. That second view is the one this chapter builds on.

Algorithm 8.1 Random-sample one-step tabular Q-planning

Input: a sample model, step size $\eta \in (0, 1]$, discount γ.

1. Initialize $Q(s, a)$ arbitrarily, with $Q(\text{terminal}, \cdot) = 0$.
2. Repeat forever:
    1. Choose a state s and an action a at random.
    2. Ask the sample model for a reward r and a next state s'.

3. $Q(s,a) \leftarrow Q(s,a) + \eta\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$.

Output: $Q \approx Q^*$ of the model.

This is Q-learning (Algorithm 6.3) with the environment replaced by the model. Under the same conditions as Q-learning, with every pair chosen infinitely often and suitably decreasing step sizes, it converges to the optimal action values *of the model* (Theorem 6.1). If the model is right, those are the optimal values of the environment.

8.1.1 Learning a model from experience

When the agent does not know the environment, it can estimate a model from its own transitions. The simplest estimate counts. If $N(s,a)$ is the number of times action a was taken in state s and $N(s,a,s')$ the number of those times it led to s', the *empirical model* is

$$\hat{P}(s' \mid s,a) = \frac{N(s,a,s')}{N(s,a)}, \qquad \hat{r}(s,a) = \text{average of the rewards that followed } (s,a). \tag{8.1}$$

Planning in the empirical model as if it were the truth is called *certainty equivalence*. In a deterministic environment one observation of each pair is enough, and the model can simply remember the last reward and next state that followed each pair; that is the model Dyna-Q uses below.

Example 8.1 A certainty-equivalence model of the slippery Trail

On the slippery Trail each go fails with probability 0.2 and leaves the hiker where she is. Suppose the agent has tried go ten times in Start and reached Middle seven times, and six times in Middle, reaching Goal five times. The empirical model says $\hat{P}(\text{Middle} \mid \text{Start}, \text{go}) = 0.7$ and $\hat{P}(\text{Goal} \mid \text{Middle}, \text{go}) = 5/6$, instead of the true 0.8 and 0.8; the exits and finish are deterministic, and their rewards are known from a single try each.

Planning in this model with $\gamma = 0.9$ gives $\hat{Q}(\text{Middle}, \text{go}) = 3.5294$ and $\hat{Q}(\text{Start}, \text{go}) = 3.0459$, against the true optimal values 3.5122 and 3.0839. The values are off by up to about 0.04, but they are good enough: the greedy policy of the model goes, and goes again, which is the optimal policy of the real Trail. Exercise 8.2 asks how bad the counts would have to be before the plan changes.

♦

An estimated model is never exact, and two things matter about its errors: how they propagate into the values, and whether the agent's own behavior lets them be corrected. The first question has a precise answer, the simulation lemma of Chapter 15 (Section 8.6.1 previews it); the second runs through the rest of this chapter.

8.2 Dyna: integrating planning, acting and learning

An agent that learns a model online can use each real transition in two ways. It can learn from the transition directly, as Q-learning does; this is *direct reinforcement learning*. And it can use the transition to improve its model, which then feeds *planning*: many updates on simulated transitions. The *Dyna architecture* (Sutton 1990) does both, in the same loop as acting: act, learn directly, learn the model, plan, and repeat. Direct learning and planning use the same update and the same value function; they differ only in where the transitions come from.

Algorithm 8.2 Tabular Dyna-Q

Input: number of planning steps $n \geq 0$, step size $\eta \in (0, 1]$, exploration rate $\varepsilon > 0$, discount γ.

1. Initialize $Q(s, a)$ arbitrarily and an empty model; $Q(\text{terminal}, \cdot) = 0$.
2. Repeat for each real step:
   1. $s \leftarrow$ the current state; choose a ε-greedily with respect to Q.
   2. Take a; observe r and s'.
   3. *Direct learning:* $Q(s, a) \leftarrow Q(s, a) + \eta\,[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$.
   4. *Model learning:* $\text{Model}(s, a) \leftarrow (r, s')$ (the environment is assumed deterministic).
   5. *Planning:* repeat n times:
      1. $\tilde{s} \leftarrow$ a random previously observed state; $\tilde{a} \leftarrow$ a random action previously taken in $\tilde{s}$.
      2. $(\tilde{r}, \tilde{s}') \leftarrow \text{Model}(\tilde{s}, \tilde{a})$.
      3. $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \eta\,[\tilde{r} + \gamma \max_{a'} Q(\tilde{s}', a') - Q(\tilde{s}, \tilde{a})]$.
   6. If s' is terminal, start a new episode.

Output: Q, and the ε-greedy policy with respect to it.

Planning samples only pairs that have actually been tried, so the model is never asked about a transition it has not seen. With $n = 0$ Dyna-Q is ordinary Q-learning. With $n > 0$ every real step is followed by n planning updates, which cost computation but no experience. In a stochastic environment the model in step 4 would store counts, as in (8.1), and step 5.2 would sample from them.

Example 8.2 Dyna-Q by hand on the Trail

Take the deterministic Trail with $\gamma = 0.9$, $\eta = 0.5$ and $Q = 0$, and let the first episode go all the way: go, go, finish, with rewards $0, 0, 4$. The direct updates leave $Q(\text{Start}, \text{go}) = Q(\text{Middle}, \text{go}) = 0$, because the next states still have value 0, and set $Q(\text{Goal}, \text{finish}) = 0.5 \cdot 4 = 2$. The model now contains the three transitions of the episode.

Suppose the next three planning updates happen to pick (Middle, go), (Start, go) and

(Goal, finish), in that order:

$$
\begin{aligned}
Q(\text{Middle}, \text{go}) &\leftarrow 0 + 0.5\,(0 + 0.9 \cdot 2 - 0) = 0.9, \\
Q(\text{Start}, \text{go}) &\leftarrow 0 + 0.5\,(0 + 0.9 \cdot 0.9 - 0) = 0.405, \\
Q(\text{Goal}, \text{finish}) &\leftarrow 2 + 0.5\,(4 - 2) = 3.
\end{aligned}
$$

Three simulated updates carried the news of the +4 back to Start. Plain Q-learning would have needed two more real episodes for that, as one-step TD did in Chapter 7. Planning is free in experience, and in a deterministic problem every simulated transition is exactly right.

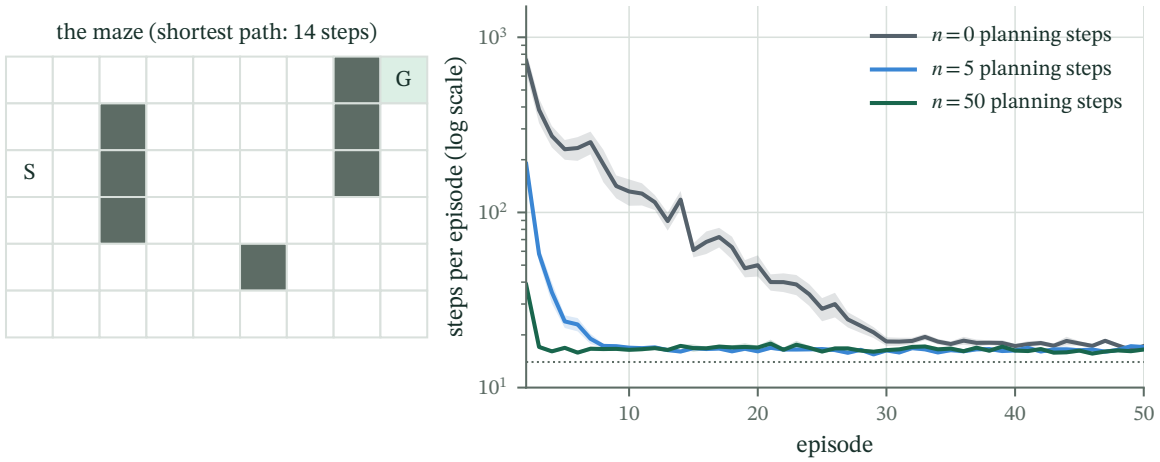


Figure 8.1. Dyna-Q in the maze of Chapter 7 (left; $\gamma = 0.95$, $\varepsilon = 0.1$, $\eta = 0.1$). Right: steps per episode, from the second episode on, for 0, 5 and 50 planning steps per real step; mean of 30 runs with a band of ± 1 standard error (log scale). The first episode is a random walk for every agent; with 50 planning steps the second episode is already close to the shortest path of 14 steps (dotted line).

Figure 8.1 shows what planning buys in the maze. The first episode is the same random walk for every agent, about 1,000 steps on average, because all estimates start equal and nothing has been learned until the goal is found. After it, the agents differ enormously. Without planning, the second episode still takes 733 steps on average (30 runs), and only after about 30 episodes does the agent walk more or less straight to the goal. With 5 planning steps per real step the second episode takes 189 steps, and with 50 it takes 39; by the third episode the 50-step agent needs 17 steps, close to the shortest path of 14 (the remaining gap is the cost of ε-greedy exploration).

The reason is the same as in Example 8.2. When the goal is first reached, only one pair learns anything from the real step. But during the planning steps that follow, the model contains the whole wandering path of the first episode, and repeated planning updates spread the value backward through it, one step per update. By the end of the first episode, or soon after, the 50-step agent has a value function that points along a good path from most of the states it has visited.

KEY IDEA

A model turns each real transition into a reusable resource. Planning with it is learning from simulated experience, with the same updates as learning from real experience, and it trades computation, which is cheap, for interaction with the environment, which is usually expensive.

8.2.1 When the model is wrong

A learned model can be wrong: it may be based on too little data, the environment may be stochastic, or the environment may change after the model was learned. Planning with a wrong model produces a policy that is optimal for the wrong world. Sometimes the error corrects itself: if the model is *optimistic*, predicting rewards that are not there, the agent goes to collect them, finds out, and fixes the model. If the model is *pessimistic*, predicting that something good is impossible, the agent never goes to look, and the error can persist forever.

Two mazes with 6 rows and 9 columns show the difference (Figure 8.2). In both, the agent starts in the bottom row, the goal is in the top-right corner, and a wall across the fourth row has a gap. In the *blocking maze* the gap is on the right; after 1,000 steps it closes, and a new gap opens on the left. The old path is blocked, the model is optimistic about it, and the agent discovers the change as soon as it walks into the new wall. In the *shortcut maze* the gap is on the left, and after 3,000 steps a second gap opens on the right, giving a shorter path (10 steps instead of 16). Here the model is pessimistic: it says the right side is a wall, so planning never sends the agent there, and only a run of exploratory moves could reveal the shortcut.

Dyna-Q+ adds a little curiosity to planning. It keeps track of the time $\tau(s, a)$ since each pair was last tried in the real environment, and in planning it replaces the modeled reward $\tilde{r}$ by

$$\tilde{r} + \kappa\sqrt{\tau(\tilde{s}, \tilde{a})}, \tag{8.2}$$

with a small constant $\kappa > 0$. Pairs that have not been tried for a long time look increasingly attractive, so the agent eventually goes back to check them. Following Sutton and Barto (2018, sec. 8.3), Dyna-Q+ also lets planning consider actions that have never been tried in an observed state, with the initial model that such an action leaves the state unchanged and earns nothing; their bonus grows like that of any other untried pair.

In the blocking maze both agents stall when the path closes, and both recover, because walking into the new wall corrects the model. Dyna-Q+ recovers faster: in the 2,000 steps after the change it collects 91 units of reward on average, Dyna-Q only 51 (it had also learned faster before the change, because its bonus explores more systematically). In the shortcut maze the difference is qualitative. Dyna-Q keeps using the long path, collecting 165 units in the 3,000 steps after the change; Dyna-Q+ finds the shortcut and collects 228.

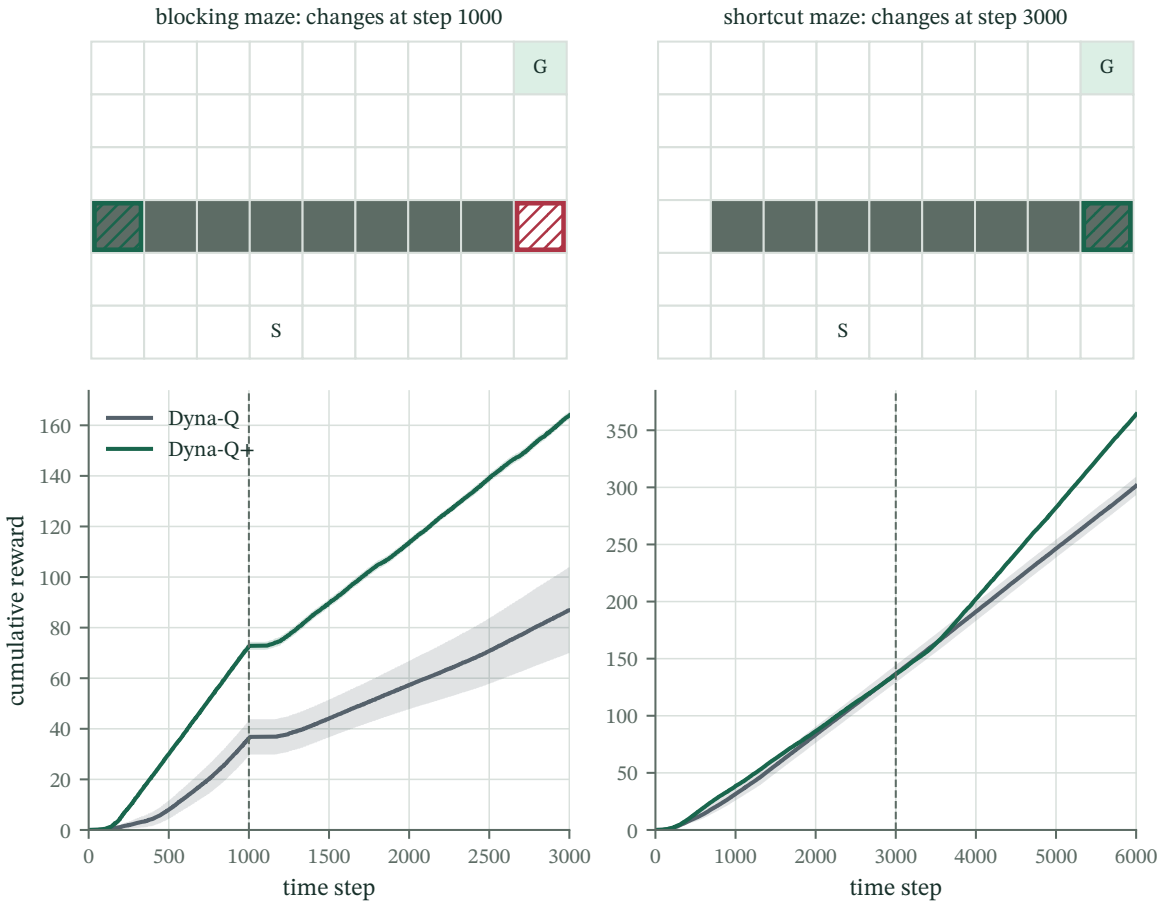


Figure 8.2. Dyna-Q and Dyna-Q+ when the environment changes. Top: the blocking maze (gap on the right; at step 1,000 the right gap closes and a left gap opens) and the shortcut maze (gap on the left; at step 3,000 a second gap opens on the right); hatched rose cells become walls, hatched green cells become open. Bottom: cumulative reward over time; mean of 10 runs with ±1 standard-error bands; 50 planning steps, $\eta = 1$, $\kappa = 10^{-3}$. Dyna-Q+ recovers faster from the blocked path and finds the shortcut, which Dyna-Q does not.

PITFALL

The exploration bonus of Dyna-Q+ needs tuning. With $\kappa = 10^{-4}$ instead of 10^{-3}, Dyna-Q+ in the shortcut maze collected no more reward after the change than Dyna-Q (in 5 runs): the bonus of the unexplored right side never outgrew the value of the known path within 3,000 steps. With κ too large, the bonus outweighs the real reward and the agent wanders off to recheck everything. Exploration in changing environments is a trade-off with no free setting (Chapter 16).

CHECKPOINT

1. What is the difference between a distribution model and a sample model? Which one does value iteration need?
2. In Dyna-Q with n planning steps, how many value updates follow each real step?
3. Why can a pessimistic model error persist forever, while an optimistic one corrects itself?

Answers.

1. A distribution model gives the probabilities of all outcomes; a sample model returns one outcome drawn with those probabilities. Value iteration needs a distribution model, because its updates are expectations over all next states.
2. $n + 1$: one direct update from the real transition and n planning updates from the model.
3. A pessimistic error makes the agent avoid the place where the model is wrong, so it never collects the evidence that would correct it. An optimistic error attracts the agent, and the real experience it then collects corrects the model.

8.3 Prioritized sweeping

Dyna-Q chooses the pairs to update uniformly at random among those it has seen. Early on, most of these updates do nothing. In the maze, after the first episode, every value is zero except near the goal, so an update of a pair far from the goal has target $0 + \gamma \cdot 0$ and changes nothing. Useful updates are those whose targets have just changed: the predecessors of states whose values have just changed. Planning should work backward from changes.

Prioritized sweeping (Moore and Atkeson 1993; Peng and Williams 1993) makes this systematic. It keeps a priority queue of pairs, ordered by how much their values would change if they were updated, which is the size of their TD error under the model:

$$P(s, a) = \left|\hat{r}(s, a) + \gamma \max_{a'} Q(\hat{s}', a') - Q(s, a)\right|, \tag{8.3}$$

where $(\hat{r}, \hat{s}')$ is the model's prediction for (s, a). Planning repeatedly updates the pair at the front of the queue. Whenever the value of a state s changes, every pair $(\bar{s}, \bar{a})$ that the model predicts will lead to s is a candidate, and it enters the queue with its own priority if that exceeds a small threshold θ.

Algorithm 8.3 Prioritized sweeping for a deterministic environment

Input: number of planning updates n per real step, step size $\eta \in (0, 1]$, threshold $\theta > 0$, exploration rate ε, discount γ.

1. Initialize $Q(s, a)$, an empty model, and an empty priority queue $\mathcal{Q}$.
2. Repeat for each real step:
   1. $s \leftarrow$ the current state; choose a ε-greedily with respect to Q; take it; observe r and s'.
   2. Model$(s, a) \leftarrow (r, s')$; record (s, a) as a predecessor of s'.
   3. $p \leftarrow |r + \gamma \max_{a'} Q(s', a') - Q(s, a)|$; if $p > \theta$, insert (s, a) into $\mathcal{Q}$ with priority p.
   4. Repeat n times, while $\mathcal{Q}$ is not empty:
      1. Remove the pair $(\tilde{s}, \tilde{a})$ of highest priority from $\mathcal{Q}$; $(\tilde{r}, \tilde{s}') \leftarrow$ Model$(\tilde{s}, \tilde{a})$.
      2. $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \eta\,[\tilde{r} + \gamma \max_{a'} Q(\tilde{s}', a') - Q(\tilde{s}, \tilde{a})]$.
      3. For each predecessor $(\bar{s}, \bar{a})$ of $\tilde{s}$, with predicted reward $\bar{r}$: $\bar{p} \leftarrow |\bar{r} + \gamma \max_{a'} Q(\tilde{s}, a') - Q(\bar{s}, \bar{a})|$; if $\bar{p} > \theta$, insert $(\bar{s}, \bar{a})$ with priority $\bar{p}$ (keeping only the higher priority if it is already in $\mathcal{Q}$).

Output: Q, and the ε-greedy policy with respect to it.

The effect is a wave of updates that sweeps backward from the goal, or from wherever something changed, through exactly the states whose values are affected. Figure 8.3 isolates planning from acting. The agent is given the full model of the maze, every value starts at zero, and we count the planning updates needed to compute the optimal values. Uniform updates of random pairs need on average 2,186 updates to bring the RMS error below 0.01 (10 runs, between 1,260 and 3,080). Prioritized sweeping, starting from the moves into the goal, needs 173, and its queue runs empty after 185 updates with the values exact: about one update per state–action pair (there are 184). On a maze twice as large in each direction, the counts are 14,029 and 737: the advantage grows with the size of the problem, because uniform updates waste a growing fraction of their effort on pairs whose values cannot change yet.

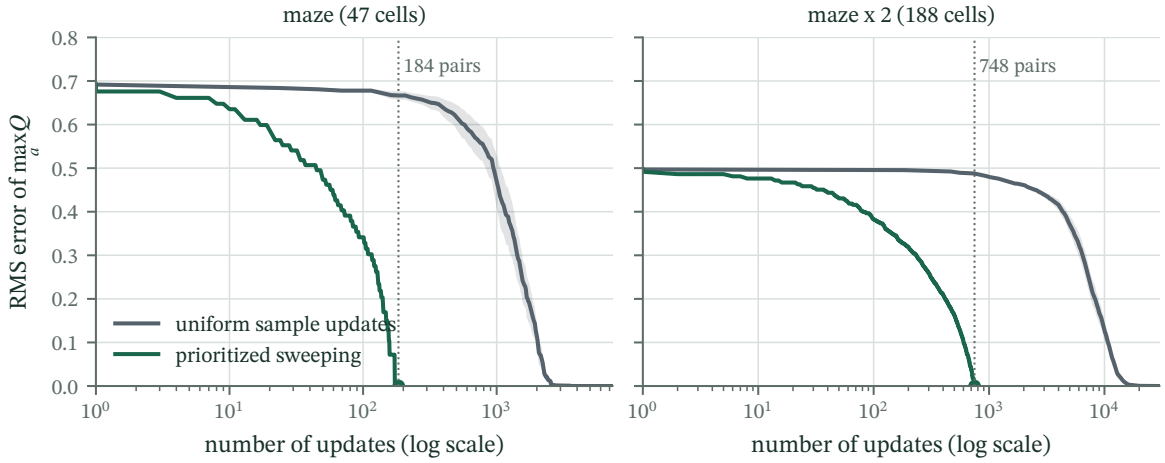


Figure 8.3. Planning with a known model of the maze (left) and of a maze twice as large in each direction (right), starting from $Q = 0$ ($\gamma = 0.95$, $\eta = 1$): RMS error of $\max_a Q$ against V^* as a function of the number of one-step updates (log scale). Gray: pairs drawn uniformly at random (10 runs, ±1 standard error); green: prioritized sweeping, which reaches the exact values (dot) after about one update per state–action pair (dotted line).

In the full agent the same focusing pays off. With 5 planning updates per real step and $\eta = 1$, prioritized sweeping in the maze needed between 14 and 26 steps for every episode after the first, using 183 planning updates in total over 15 episodes (one run); the queue is empty most of the time, so most of the allowed planning steps are not even needed.

In a stochastic environment the model must store estimated probabilities, the update in step 4.2 becomes an expected update over the possible next states, and a predecessor's priority is scaled by the probability with which it leads to the changed state (Exercise 8.13). Prioritized sweeping is one instance of a general principle that recurs in this chapter: planning effort should go where it can change the values that matter. Prioritized experience replay in deep Q-networks (Section 10.6) applies the same principle to a buffer of real transitions.

8.4 Expected versus sample updates

Every value update in this book can be described by three choices: whether it updates state values or action values, whether it estimates the values of a given policy or the optimal values, and whether it is an *expected update*, which averages over all possible next states with a distribution model, or a *sample update*, which uses one sampled next state. Table 8.1 places the methods of earlier chapters in this scheme.

Table 8.1. One-step updates by the value they estimate and by how they use the model.

value estimated	expected update (distribution model)	sample update (one transition)
V^π	policy evaluation (Algorithm 4.1)	TD(0) (Algorithm 6.1)
V^*	value iteration (Algorithm 4.3)	(none in this book)
Q^π	Q-policy evaluation	SARSA (Algorithm 6.2)
Q^*	Q-value iteration	Q-learning (Algorithm 6.3)

Consider the update of Q^* for one pair (s, a). The expected update computes

$$Q(s,a) \leftarrow \sum_{s'} P(s' \mid s,a)\big[r(s,a) + \gamma \max_{a'} Q(s',a')\big], \tag{8.4}$$

and the sample update draws one s' and moves toward it, $Q(s,a) \leftarrow Q(s,a) + \eta[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$. If the next-state values are correct, the expected update is exact after one application, while the sample update is noisy and needs many applications with decreasing step sizes. But the expected update costs more. If (s,a) has b possible next states (its *branching factor*), the expected update evaluates $\max_{a'} Q(s',a')$ for all b of them, which takes roughly b times the computation of one sample update.

Which is better for a fixed amount of computation? Suppose the b next states are equally likely, their values are correct, the initial estimate has error 1, and the values of the next states are

spread like independent draws with variance 1. The expected update reduces the error to zero, but only after all b terms have been computed. Sample updates with step sizes $\eta_t = 1/t$ average the first t sampled targets, and their root-mean-square error after t updates is

$$\sqrt{\frac{b-1}{b\,t}} \tag{8.5}$$

(Exercise 8.11 derives it). Figure 8.4 plots both against computation. For large b, a small fraction of the cost of one expected update brings the sample estimate close to the truth: with $b = 1{,}000$, after 100 sample updates, a tenth of the cost of an expected update, the error is already about 0.1. And in a real planning problem the next-state values are themselves estimates, so the precision of an expected update is partly wasted; it is usually better to spend the computation on many pairs, roughly right, than on a few, exactly.

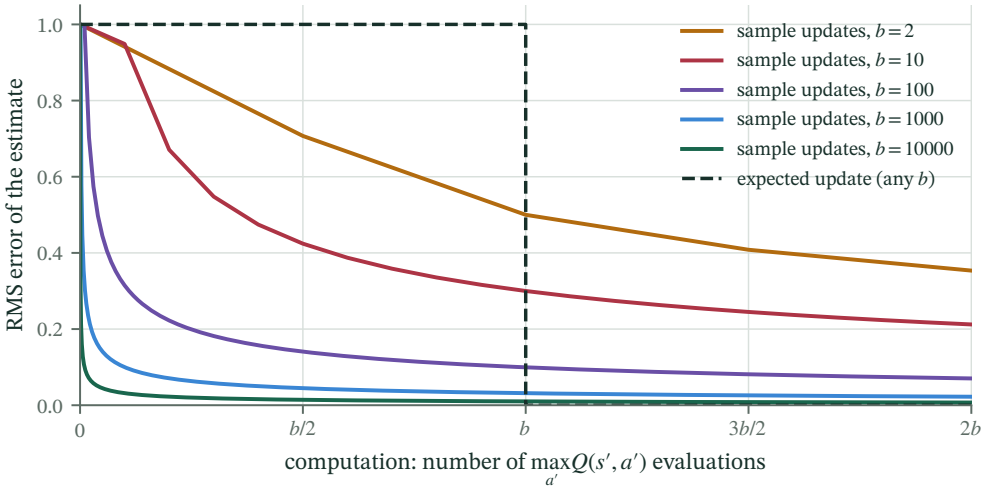


Figure 8.4. Expected versus sample updates for one state–action pair with b equally likely successors whose values are correct: RMS error of the estimate as a function of computation, measured in evaluations of $\max_{a'} Q(s', a')$. The expected update (dashed) is exact after b evaluations and useless before; sample updates (step sizes $1/t$) improve immediately, and for large branching factors they reach a small error long before an expected update could finish.

8.4.1 Trajectory sampling

The second question is where to make the updates. Dynamic programming sweeps through all states, giving each the same attention. But in a large problem most states are irrelevant: they are reached rarely, or only by bad policies. An alternative is *trajectory sampling*: simulate episodes with the model, following the current policy, and update the states along the way. Updates are then distributed according to the *on-policy distribution*, the distribution of states the policy actually visits. This focuses computation on the states that matter for the current policy, at the risk of neglecting states that the policy wrongly avoids.

Figure 8.5 repeats an experiment of Sutton and Barto (2018, sec. 8.6) on random tasks. Each task has 1,000 or 10,000 states and two actions; each pair leads to one of b randomly chosen next states with equal probability, with a reward drawn from a standard normal distribution, and

every transition ends the episode with probability 0.1. Both methods make expected updates of Q^*; the uniform method cycles through all pairs, and the on-policy method updates the pairs visited by ε-greedy episodes ($\varepsilon = 0.1$) from the start state. The plot shows the true value, under the greedy policy, of the start state.

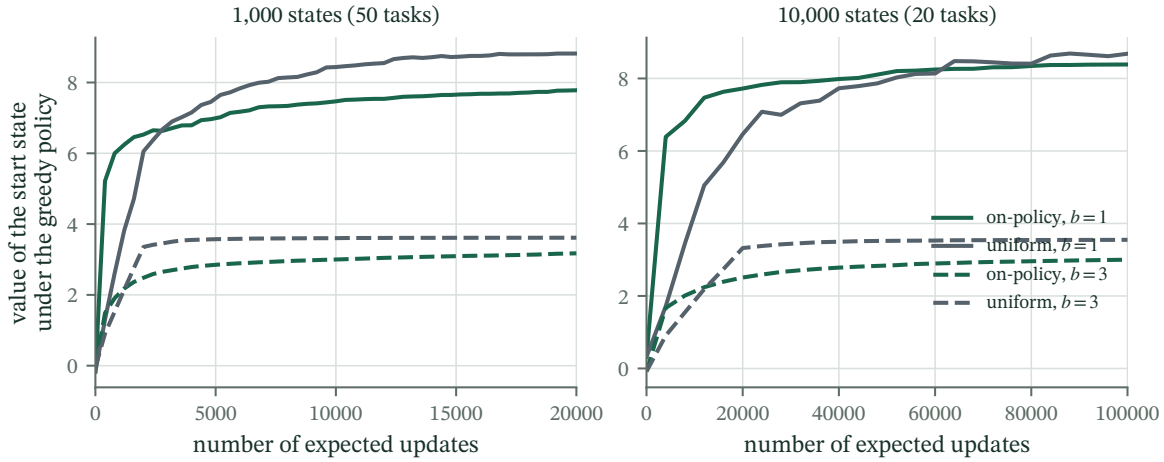


Figure 8.5. Trajectory sampling on random tasks with 1,000 states (left; 50 tasks) and 10,000 states (right; 20 tasks), two actions and branching factor $b = 1$ (solid) or $b = 3$ (dashed). Updates distributed along on-policy trajectories (green) improve the start state's value much faster at first; sweeping uniformly (gray) catches up and ends higher. The on-policy advantage lasts longer with more states and with fewer branches.

On-policy sampling wins early: with 1,000 states and $b = 1$, after 400 updates the greedy policy's start value is 5.22 for on-policy sampling against 1.31 for uniform sweeps. The uniform method catches up after about 2,800 updates ($b = 1$) or 1,600 ($b = 3$) and ends higher, because the on-policy method keeps updating states whose values are already good and neglects others. With 10,000 states the on-policy advantage lasts far longer, until about 64,000 updates for $b = 1$ and 16,000 for $b = 3$. These match the conclusions of Sutton and Barto (2018): on-policy sampling helps most in large problems with few branches, exactly where uniform sweeps waste the most effort.

8.4.2 Real-time dynamic programming

Real-time dynamic programming (RTDP) (Barto et al. 1995) is trajectory sampling taken to its logical end: on-policy trajectories with expected updates of V^*. At each step of a trial it applies the value-iteration update to the current state only, acts greedily with respect to the updated values, and moves on. It is asynchronous value iteration (Section 4.6) in which the states to update are chosen by the agent's own greedy behavior.

Algorithm 8.4 Real-time dynamic programming (trial-based RTDP)

Input: a distribution model (P, r) of an episodic task, a start state, initial values V_0 with $V_0 \geq V^*$ and $V_0(\text{goal}) = 0$, discount $\gamma \leq 1$.

1. $V \leftarrow V_0$.
2. For each trial:
   1. $s \leftarrow$ the start state.
   2. Repeat until s is a goal state:
      1. For each action a: $Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V(s')$.
      2. $V(s) \leftarrow \max_a Q(s,a)$.
      3. Take a greedy action $a \in \arg\max_a Q(s,a)$ (in the real environment, or in the model) and observe the next state s'; $s \leftarrow s'$.

Output: V, and the greedy policy with respect to it.

RTDP is designed for *stochastic shortest-path problems*: episodic tasks in which every step costs something (a negative reward) and the goal is to reach a terminal goal state as cheaply as possible. For these problems it converges to optimal behavior without visiting every state.

Theorem 8.2 Convergence of RTDP (Barto, Bradtke and Singh, 1995)

Consider an undiscounted episodic task with finitely many states in which (1) the initial value of every goal state is zero, (2) there is at least one policy that reaches a goal state with probability one from every start state, (3) every reward for a transition from a non-goal state is strictly negative, and (4) every initial value is at least as large as its optimal value. Then, with probability one, RTDP converges to a policy that is optimal on all relevant states, the states reachable from the start states under some optimal policy, and to the optimal values there. Irrelevant states may be visited only finitely often, or never.

The theorem is proved by Barto et al. (1995); we follow the statement of Sutton and Barto (2018, sec. 8.7). Condition (4) is the key: values start *optimistic*, so a state that looks better than it is attracts the agent, gets updated, and loses its false promise, while a state that is truly good is never undervalued, so the greedy policy cannot lock onto a bad route for lack of information (Exercise 8.9 shows what goes wrong with pessimistic values). With negative rewards, $V_0 = 0$ satisfies it. So does any *admissible heuristic*, an estimate that never underestimates the value, such as minus the straight-line distance to the goal, and a good heuristic focuses RTDP further. RTDP generalizes the learning real-time A* algorithm of heuristic search (Korf 1990) to stochastic problems.

Consider an open room of 15 by 15 cells, with the start in the middle of the left wall and the goal in the middle of the right wall. Every move costs 1, and with probability 0.2 a move slips to one of the two perpendicular directions; the optimal expected cost from the start is 19.41. Starting from $V_0 = 0$, RTDP updates all 224 non-goal states in its first 20 trials, because unexplored states look attractive, and after 100 trials its estimate of the start state is -18.36. Starting from the Manhattan-distance heuristic, which is admissible here because no move gains more than one unit of distance, it updates only 94 states in 20 trials, and its greedy policy already reaches the goal in about 19.5 steps on average, essentially optimal (the optimum is 19.41); after 100 trials it

has still touched only 141 states. RTDP spends its computation along the routes the task actually uses.

8.5 Decision-time planning

All the planning so far is *background planning*: it improves a value function or policy for all the states the agent might meet, and when a state arrives, the agent reads off the action. *Decision-time planning* starts from the state the agent is actually in and computes an action for that state alone, and usually discards most of the computation afterward. It is the planning a chess player does when thinking about a move. It pays off when the agent has time to think before acting (as in board games) and when the states are too many to plan for in advance.

8.5.1 Heuristic search

The classic form is *heuristic search*: build a tree of the possible continuations from the current state to some depth, evaluate the leaves with a value function (the *heuristic evaluation*), back the values up to the root with maximizing (or, in games, minimax) updates, and choose the best action at the root. A deeper search uses more computation to correct the errors of an imperfect value function, in the spirit of the error-reduction property of Chapter 7: the error of the leaf values is discounted by the depth. Game-playing programs have combined learned evaluations with search for a long time; the later versions of TD-Gammon added a few moves of lookahead search to their learned evaluation function (Tesauro 1995). The search algorithm A* (Hart et al. 1968) is the classic method for deterministic shortest paths with an admissible heuristic.

8.5.2 Rollout algorithms

A *rollout algorithm* (Tesauro and Galperin 1996) estimates, at decision time, the value of each action by simulation. For every action a available in the current state s, it runs many simulated episodes that start with a and then follow a fixed *rollout policy* (or base policy) π, and it averages their returns. These are Monte Carlo estimates of $Q^\pi(s, a)$ (Chapter 5). The agent then takes the action with the highest estimate.

This is one step of policy improvement (Theorem 4.6): the policy that acts greedily with respect to Q^π is at least as good as π, so a rollout algorithm improves on its base policy, whatever the base policy is. The computation is easy to parallelize, and the estimates need only be good enough to *rank* the actions. Tesauro and Galperin (1996) used rollouts with a learned backgammon program as the base policy and reported a large reduction in its error rate.

Example 8.3 Rollouts on the Trail

Let the rollout policy be uniform. From Start, every rollout that begins with exit returns 1. Rollouts that begin with go return 1.8 (if the base policy then exits at Middle) or 3.24 (if it goes on), each with probability one half; 5,000 of them averaged 2.525, close to $Q^\pi(\text{Start}, \text{go}) = 2.52$. The rollout algorithm therefore chooses go, and in Middle, where the estimates are 2 and 3.6,

it again chooses go. One step of improvement on the uniform policy gives the optimal policy here. Even a single rollout per action would do, since both possible returns after go exceed 1.

♦

8.5.3 Monte Carlo tree search

Rollout algorithms treat every action at the root separately and never learn from one simulation where to look in the next. *Monte Carlo tree search* (MCTS) (Coulom 2006; Kocsis and Szepesvári 2006; Browne et al. 2012) keeps the statistics of its simulations in a search tree rooted at the current state, and grows the tree toward the most promising continuations. Each iteration has four phases (Figure 8.6).

1. Selection. Starting at the root, descend the tree by choosing at each node a child according to a *tree policy* that balances the estimated value of each child against its uncertainty, until reaching a node with an untried move.
2. Expansion. Add a child for one untried move.
3. Simulation. From the new node, play the game or episode to the end with a *rollout policy*, often uniformly random moves.
4. Backup. Pass the result back up the path: every node on it increments its visit count N and adds the result to its total value W.

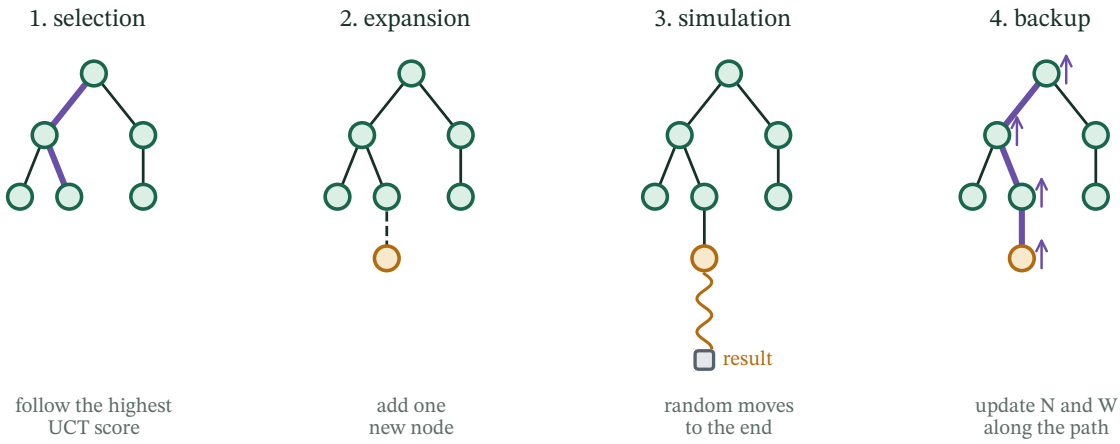


Figure 8.6. The four phases of one iteration of Monte Carlo tree search. Selection follows the highest UCT scores from the root to a node with an untried move; expansion adds one node; simulation plays random moves to the end of the game; backup updates the visit counts and values of every node on the path.

The most influential tree policy is *UCT* (upper confidence bounds applied to trees) (Kocsis and Szepesvári 2006), which treats the choice at each node as a bandit problem and applies the UCB1 rule of Algorithm 2.2. At a node visited N times, it selects the child j that maximizes

$$\frac{W_j}{N_j} + c\sqrt{\frac{\ln N}{N_j}}, \tag{8.6}$$

where N_j is the number of visits to child j, W_j the total result of the simulations through it (from the point of view of the player who chooses at the node), and c an exploration constant; children not yet visited are tried first. The first term favors moves that have done well; the second favors moves that have been tried rarely compared with their parent. With results in $[0, 1]$, $c = \sqrt{2}$ gives exactly UCB1.

Algorithm 8.5 Monte Carlo tree search with UCT

Input: the current state s_0, a number of iterations, exploration constant c, a rollout policy; a model that gives the moves available in a state, their results, and the outcome at the end of a game.

1. Create a root node for s_0 with $N = 0$.
2. Repeat for each iteration:
   1. *Selection:* $v \leftarrow$ root; while v has no untried moves and has children: $v \leftarrow$ the child of v that maximizes (8.6).
   2. *Expansion:* if v has an untried move, create a child of v for one of them and set v to it.
   3. *Simulation:* from the state of v, follow the rollout policy to the end; let z be the outcome.
   4. *Backup:* for each node u on the path from v to the root: $N_u \leftarrow N_u + 1$; $W_u \leftarrow W_u + z_u$, where z_u is the outcome for the player who made the move into u.
3. Output: the move of the root's most visited child.

The tree grows unevenly. Promising moves are visited often, so their subtrees grow deep, while poor moves are revisited only occasionally, as their exploration bonus slowly grows. As the number of iterations grows, the values near the root approach their minimax or optimal values, and the most visited move becomes the best one (Section 8.6.3). In a single-agent problem, the same algorithm uses the return of the simulation as the result. On the Trail, scaled so that results lie in $[0, 1]$, 2,000 iterations visited go 1,962 times, with an average return of 3.18 (approaching $Q^*(\text{Start}, \text{go}) = 3.24$ as the tree learns to go on at Middle), and exit 38 times.

Example 8.4 One UCT selection

A node has been visited $N = 20$ times. Its three children have $(N_j, W_j) = (10, 6)$, $(6, 4.2)$ and $(4, 1.6)$, so their average results are 0.6, 0.7 and 0.4. With $c = \sqrt{2}$, the exploration terms are $\sqrt{2 \ln 20 / N_j} = 0.774$, 0.999 and 1.224, and the scores are

$$0.6 + 0.774 = 1.374, \qquad 0.7 + 0.999 = 1.699, \qquad 0.4 + 1.224 = 1.624.$$

UCT selects the second child: it has the best average, and few enough visits to keep a large bonus. The third child, with the worst average, comes a close second, because it has been tried only four times; one bad result for the second child would make the third one next (Exercise 8.6).

♦

Example 8.5 Tree search in Nim

In *Nim*, two players take turns removing one or more objects from one of several heaps; whoever takes the last object wins. Nim is solved: the player to move loses with perfect play exactly when the bitwise exclusive-or (XOR) of the heap sizes is zero, and otherwise the winning moves are those that make it zero. From heaps $(1, 2, 4)$, whose XOR is 7, the only winning move takes one object from the heap of 4, leaving $(1, 2, 3)$, whose XOR is 0.

UCT with uniformly random rollouts finds the winning move from small positions: in our runs it found it in all 20 of 20 runs from $(2, 3)$ and $(1, 3)$ with 300 iterations, from $(1, 2, 4)$ with 1,000, and from $(3, 5)$ and $(2, 3, 4)$ with 3,000. From $(3, 4, 5)$, whose only winning move takes two objects from the heap of 3, it failed: after 50,000 iterations the most visited move was a losing one, and the average results of the five most visited moves were all between 0.48 and 0.52. The reason is the rollouts. Under uniformly random play, each of the twelve moves from $(3, 4, 5)$ wins with probability exactly $\frac{1}{2}$ (an exact calculation, Exercise 8.14), so the simulations carry no information about which move is right, and UCT can distinguish the moves only by growing a tree deep enough to see the game's end, which takes far more iterations.

♦

PITFALL

MCTS is only as good as the information in its simulations. When random playouts are uninformative, as in Nim, or misleading, as in positions where a single precise move decides the game, UCT needs enormous trees. Strong programs therefore replace random rollouts by better rollout policies, or by learned value functions that evaluate the leaves directly: AlphaGo combined MCTS with policy and value networks (Silver et al. 2016), and AlphaZero dropped rollouts entirely (Silver et al. 2018).

In tic-tac-toe, by contrast, random playouts are informative enough. In our runs UCT won 15, drew 4 and lost 1 of 20 games against a random player with 50 iterations per move, won 18 and drew 2 with 200, and won 19 and drew 1 with 1,000; two UCT players with 1,000 iterations each drew all 10 of their games, which is the result of perfect play.

MCTS is the planning engine of AlphaGo, AlphaGo Zero and AlphaZero (Silver et al. 2016, 2018; Silver, Schrittwieser, et al. 2017), which use a variant of UCT in which a learned policy network supplies prior probabilities for the moves and a value network evaluates the new nodes, and of MuZero (Schrittwieser et al. 2020), which searches with a learned model. Section 15.7 describes them.

LAB

Lab 8.1: Planning with models (`labs/ch08_planning.py`, about 45 minutes). Run `python3 labs/ch08_planning.py`. You will reproduce Example 8.2, compare Dyna-Q with 0, 5 and 50 planning steps in the maze, watch Dyna-Q+ find the shortcut, count the planning updates of prioritized sweeping and of uniform planning, run RTDP with and without a heuristic, and plan at decision time with rollouts on the Trail and with UCT in Nim and tic-tac-toe.

8.6 The theory of planning with sampled models ●●

8.6.1 Planning in an estimated model

Certainty equivalence (Section 8.1.1) is consistent. If every pair is tried infinitely often, the empirical probabilities and rewards (8.1) converge to the true ones by the law of large numbers, and the optimal values of an MDP depend continuously on its transition probabilities and rewards, so the values computed in the empirical model converge to V^*, and greedy policies become optimal once the errors fall below half the gap between the best and second-best action values. The quantitative version is the *simulation lemma* of Chapter 15 (Lemma 15.1): if the estimated transition probabilities of every pair are within ϵ_P of the true ones (in total variation or ℓ_1 distance) and the estimated rewards within ϵ_r, then the value of every policy in the model is within $O\big(\gamma\epsilon_P r_{\max}/(1-\gamma)^2 + \epsilon_r/(1-\gamma)\big)$ of its true value, where $r_{\max}$ bounds the rewards. The factor $1/(1-\gamma)^2$ reflects errors that compound over an effective horizon of $1/(1-\gamma)$ steps. Chapter 15 proves the lemma and discusses the sample complexity of certainty-equivalence planning, which turns out to be close to optimal when data can be generated for every pair.

The statistical guarantee is not the whole story. Certainty equivalence says nothing about *which* data the agent collects: an agent that plans greedily in its empirical model may never try the actions that would correct it, as the shortcut maze showed. Combining planning with deliberate exploration, for example by planning in an optimistic model, is the subject of Chapter 16.

8.6.2 Sparse sampling

Can planning with a sample model avoid the dependence on the number of states altogether? *Sparse sampling* (Kearns et al. 2002) shows that it can, at a price. To choose an action in state s, it builds a tree of depth H: for every action it draws C next states from the sample model, and from each of those it repeats the procedure, down to depth H. It then computes the values in the tree by expected updates over the C samples (averages instead of probabilities) and maximization over actions, and acts greedily at the root.

Theorem 8.3 Sparse sampling (Kearns, Mansour and Ng, 2002)

For a discounted MDP with $|\mathcal{A}|$ actions and rewards in $[0, r_{\max}]$, and any $\epsilon > 0$, there are a depth H and a width C that depend only on ϵ, γ, $r_{\max}$ and $|\mathcal{A}|$, not on the number of states, such that the sparse-sampling policy is ϵ-optimal from every state. The depth is $H = O\big(\frac{1}{1-\gamma}\log\frac{r_{\max}}{\epsilon(1-\gamma)}\big)$, the width is polynomial in $r_{\max}/(\epsilon(1-\gamma))$, H and $\log|\mathcal{A}|$, and each decision uses $O((|\mathcal{A}|C)^H)$ calls to the sample model.

The result is remarkable in both directions. Its cost does not depend on the size of the state space at all, so it applies in principle to problems with infinitely many states. But the cost is exponential in the effective horizon: the tree has $\sum_{d=1}^{H}(|\mathcal{A}|C)^d$ leaves and internal nodes, which is already 3,368,420 model calls for two actions, $C = 10$ and $H = 5$ (Exercise 8.8). Sparse sampling is a statement about what is possible rather than a practical algorithm. UCT and the other forms of

MCTS can be seen as attempts to spend the samples adaptively, deepening the tree only where it matters, instead of uniformly.

8.6.3 What UCT guarantees

The bandits at the nodes of a search tree are not the stationary bandits of Chapter 2: the average result of a child changes as its own subtree grows and its choices improve. Kocsis and Szepesvári (2006) analyzed UCB1 under such drifting payoffs and proved that UCT is consistent.

Theorem 8.4 Consistency of UCT (Kocsis and Szepesvári, 2006)

Consider a finite-horizon MDP or game tree of depth D with K actions per node and results in $[0, 1]$, searched by UCT with an exploration constant suitably scaled with the depth. Then the bias of the estimated value of the root is $O(\log n/n)$ after n iterations, and the probability that UCT selects a suboptimal action at the root converges to zero at a polynomial rate as $n \to \infty$.

Consistency is an asymptotic guarantee, and the asymptotics can be slow. Since every node keeps exploring, a branch that looks poor in its early simulations is still revisited, but only logarithmically often; if its value is hidden deep in the tree, UCT can take a very long time to find it. Coquelin and Munos (2007) constructed trees of depth D on which the regret of UCT grows like an iterated exponential in D, and proposed modified bonuses with better worst-case guarantees. The Nim position of Example 8.5 is a milder version of the same phenomenon. In practice, the success of MCTS depends less on the exact bonus than on the quality of the rollout policy or leaf evaluation, and modern programs use learned ones.

8.7 From tables to deep models ●●●

Tabular Dyna works because a table model is exact for every transition it has seen, and because every simulated update concerns a real pair. Neither property survives in large problems. A model that generalizes, such as a neural network that predicts the next state from the current one, can be wrong about pairs that look like the ones it was trained on; planning then exploits its errors, steering the policy toward states where the model is optimistic. Errors also compound: a simulated trajectory of k steps feeds the model its own predictions, and small one-step errors grow with k. And in high-dimensional observation spaces, predicting the next observation in detail, every pixel, wastes capacity on things irrelevant to the task.

Several lines of work respond to these problems; Chapter 15 treats them in detail (Section 15.1). One observation is that a replay buffer of real transitions (Section 10.2.1) is itself a nonparametric model, exact on the transitions it stores: replaying it, as DQN does, is a form of Dyna without generalization, and Hasselt et al. (2019) argue that a parametric model pays off only when it generalizes or plans in ways a replay buffer cannot. Model-based policy optimization keeps model rollouts short, branching from real states, to limit compounding errors (Janner et al. 2019) (Algorithm 15.2). Latent world models such as Dreamer learn compact state representations in which they imagine whole trajectories and train an actor-critic entirely in imagination (Hafner

et al. 2020) (Section 15.6). And MuZero learns a model that predicts only what tree search needs, rewards, values and policies, rather than observations (Schrittwieser et al. 2020) (Section 15.7).

RESEARCH NOTE

Open questions. What makes a learned model useful for planning, as opposed to accurate in prediction: should models be trained to predict observations, or only the quantities that planning uses? How should an agent decide how much to trust its model in each part of the state space, and how long its simulated rollouts should be? When does planning with a learned model beat model-free learning with the same data and computation? And how should planning effort be allocated, the question of prioritized sweeping and trajectory sampling, when the model and the value function are both neural networks? Chapter 15 and Chapter 24 return to these questions.

8.8 Summary

- A model predicts the next state and reward. Distribution models give all probabilities and support expected updates; sample models give one outcome and support sample updates. A model can be estimated from experience by counting (certainty equivalence).
- Planning is learning from simulated experience: the same updates as in learning, applied to transitions from a model. Dyna-Q interleaves acting, direct Q-learning, model learning and planning; with 50 planning steps per real step it walked the maze almost optimally after two episodes, where Q-learning needed about 30.
- Model errors matter in different ways. Optimistic errors correct themselves; pessimistic ones, such as an unknown shortcut, can persist forever. Dyna-Q+ adds an exploration bonus $\kappa\sqrt{\tau}$ for transitions not tried recently.
- Prioritized sweeping focuses planning on the pairs whose values would change most, working backward from changes; with a known model of the maze it computed the optimal values in about one update per state–action pair, more than ten times fewer than uniform updates.
- Expected updates are exact but cost about b times more than sample updates; with large branching factors, sample updates give most of the benefit at a fraction of the cost. Trajectory sampling concentrates updates on the states the policy visits; it is faster early and in large problems, while uniform sweeps are better in the long run.
- RTDP applies value-iteration updates along greedy trajectories; with optimistic initial values it converges to optimal behavior on the relevant states without visiting every state, and admissible heuristics focus it further.
- Decision-time planning computes the action for the current state: heuristic search, rollout algorithms (one step of policy improvement by simulation) and Monte Carlo tree search, whose UCT rule treats each node as a bandit. MCTS is the engine of AlphaGo, AlphaZero and MuZero, but it is only as good as the information in its simulations.
- (●●) Planning in an estimated model is consistent, with errors controlled by the simulation lemma; sparse sampling plans at a cost independent of the number of states but exponential in the horizon; UCT is consistent, but can be very slow in the worst case.

The tabular methods of Parts I and II end here. Part III replaces tables by function approximators,

which raises new problems for every method of this chapter and of the previous ones, beginning with value function approximation in Chapter 9.

8.9 Further Reading

Sutton and Barto (2018), Chapter 8, is the standard treatment and the source of several of the experiments repeated here (the Dyna maze, the blocking and shortcut mazes, expected versus sample updates and trajectory sampling). Sutton (1990) introduced the Dyna architecture. Prioritized sweeping was developed at the same time by Moore and Atkeson (1993) and Peng and Williams (1993); both papers are short and clear. Barto et al. (1995) introduced real-time dynamic programming and connected reinforcement learning with heuristic search; Korf (1990) is the classic paper on real-time heuristic search, and Hart et al. (1968) introduced A*.

For decision-time planning, Tesauro and Galperin (1996) introduced rollout algorithms in backgammon, Coulom (2006) coined the name Monte Carlo tree search, and Kocsis and Szepesvári (2006) introduced UCT with its consistency analysis. Browne et al. (2012) is a comprehensive survey of MCTS variants and applications. Bertsekas (2019) treats rollout and approximate policy iteration from the viewpoint of optimal control. The landmark programs that combine search with learning are described in Silver et al. (2016), Silver, Schrittwieser, et al. (2017), Silver et al. (2018) and Schrittwieser et al. (2020).

(●●) Kearns et al. (2002) is the original sparse-sampling paper; its analysis is short and instructive. Coquelin and Munos (2007) study the worst case of UCT and propose alternatives. The sample complexity of planning in estimated models, and the simulation lemma, are treated in Chapter 15 and in Agarwal et al. (2022).

(●●●) Hasselt et al. (2019) discuss when parametric models are worth more than replay. For deep model-based reinforcement learning, start with Janner et al. (2019) and Hafner et al. (2020), and then Chapter 15.

8.10 Exercises

Exercise 8.1 ● Dyna updates by hand

On the deterministic Trail with $\gamma = 0.9$, $\eta = 0.5$ and $Q = 0$, the first real episode is go, exit (rewards 0 and 2). (a) Give the direct Q-learning updates and the contents of the model after the episode. (b) Compute the values after three planning updates of (Start, go), (Middle, exit) and (Start, go), in that order. (c) If planning continues forever, what values does it converge to, and why is Q(Middle, go) not among them?

Exercise 8.2 ● A certainty-equivalence model

Use the data of Example 8.1. (a) Write down the empirical model of the slippery Trail. (b) Verify that $\hat{Q}(\text{Middle}, \text{go}) = 3.5294$ and $\hat{Q}(\text{Start}, \text{go}) = 3.0459$ by solving the Bellman optimality

equations of the model. (c) Let $\hat{p}$ be the estimated success probability of go in Middle. For which values of $\hat{p}$ does the plan exit at Middle?

Exercise 8.3 ● The Dyna-Q+ bonus

(a) With $\kappa = 10^{-3}$, compute the bonus of a pair that was last tried 100, 1,000 and 10,000 steps ago. (b) In the shortcut maze with $\gamma = 0.95$ and reward 1 at the goal, compare the bonus after 3,000 steps with the difference between the values of the start state along the long path (16 steps) and along the shortcut (10 steps). (c) Explain why Dyna-Q, without a bonus, is unlikely to find the shortcut. (d) What goes wrong if κ is much too small or much too large?

Exercise 8.4 ● The order of prioritized sweeping

A chain of four states $s_1 \to s_2 \to s_3 \to s_4 \to$ goal has one action per state, reward 1 for entering the goal and 0 otherwise, and $\gamma = 0.9$. The model is known, $Q = 0$, and $\eta = 1$. (a) List the updates that prioritized sweeping makes, with their priorities, until its queue is empty, and the final values. (b) If instead each planning update picks one of the four states uniformly at random, what is the expected number of updates until $Q(s_1)$ is correct?

Exercise 8.5 ● Expected or sample?

(a) A pair has $b = 1{,}000$ equally likely successors. After 100 sample updates with step sizes $1/t$, what RMS error does (8.5) predict? How far has an expected update got with the same computation? (b) For $b = 2$, compare the two after two evaluations of $\max_{a'} Q(s', a')$. (c) Explain why, in a large planning problem, spreading sample updates over many pairs can beat exact expected updates of a few.

Exercise 8.6 ● UCT selections

Continue Example 8.4. (a) The simulation through the selected child ends in a loss (result 0). Update the statistics and compute the three UCT scores at the next visit of the node. Which child is selected? (b) How do the scores change if the exploration constant is $c = 0.5$ instead of $\sqrt{2}$? (c) Explain in words why a child that is never selected must eventually be selected again.

Exercise 8.7 ● Rollout policies on the Trail

(a) Run the rollout algorithm of Example 8.3 with the base policy "always exit". Which actions does it choose in Start and Middle? (b) Is the resulting policy optimal? Relate your answer to the policy improvement theorem. (c) Give an example of a base policy whose rollout policy is not optimal on the Trail with $\gamma = 0.9$, or explain why none exists.

Exercise 8.8 ●● The cost of sparse sampling

(a) Show that a sparse-sampling tree of depth H and width C with $|\mathcal{A}|$ actions makes $\sum_{d=1}^{H}(|\mathcal{A}|C)^d$ calls to the sample model when no episode ends inside the tree, and evaluate it for $|\mathcal{A}| = 2$, $C = 10$, $H = 5$, and for $|\mathcal{A}| = 4$, $C = 20$, $H = 10$. (b) To make the discounted value beyond depth H smaller than ϵ when rewards lie in $[0, 1]$, how large must H be? Evaluate it for $\gamma = 0.9$ and $\epsilon = 0.1$. (c) Compare with the cost of one sweep of value iteration on a problem with $|\mathcal{S}|$ states, and say when each is preferable.

Exercise 8.9 ●● Why RTDP needs optimism

(a) Construct a small shortest-path problem with two routes from the start, in which RTDP with pessimistic initial values ($V_0 < V^*$ at some state) converges to a suboptimal route with probability one. (b) Show that with $V_0 \geq V^*$ everywhere, the values RTDP computes remain at least V^* at every step. (c) Use (b) to explain informally why the greedy policy cannot settle on a route that is worse than optimal.

Exercise 8.10 ●● Planning with a wrong reward

An agent's model of the Trail believes that finish pays 2 instead of 4 (say from a noisy first observation). (a) Compute the optimal policy of the model with $\gamma = 0.9$ and its true value. (b) If the agent acts greedily and plans with Dyna, will the error ever be corrected? (c) Name two changes to the agent that would correct it, and say what each costs.

Exercise 8.11 ●● The error of sample updates

Derive (8.5). Assume that the b successor values $v_1, \dots, v_b$ are independent with variance 1, that the target is their average $\bar{v}$, and that sample updates with step sizes $1/t$ average t independent uniform draws from $\{v_1, \dots, v_b\}$. (Hint: condition on the v_i first.)

Exercise 8.12 ●● Tree search for tic-tac-toe

Using `rlcourse.planning.uct_search` and `TicTacToe`: (a) play 20 games of UCT against a random player for 50, 200 and 1,000 iterations per move, alternating who moves first, and report wins, draws and losses; (b) play UCT against itself with 1,000 iterations per move; (c) find a position in which UCT with 50 iterations misses an immediate win or fails to block an immediate loss, and explain why.

Exercise 8.13 ●● Prioritized sweeping with stochastic transitions

Extend Algorithm 8.3 to environments with stochastic transitions. (a) What must the model store, and what does the update of a pair become? (b) When the value $\max_a Q(s, a)$ of a state changes by Δ, by how much can the target of a predecessor pair $(\bar{s}, \bar{a})$ change? Use this to define its priority. (c) Why can expected updates be wasteful here, and what would a sample-based version of prioritized sweeping look like?

Exercise 8.14 ●●● When rollouts carry no information

(a) Let $p(h)$ be the probability that the player to move wins a game of Nim from heaps h when both players move uniformly at random. Write a recursion for p, compute it for every move from $(3, 4, 5)$, and confirm that each move wins with probability $\frac{1}{2}$ for the mover. (b) Explain why UCT with random rollouts struggles from this position, while it solves $(1, 2, 4)$ quickly. (c) Discuss three remedies, such as better rollout policies, learned evaluations, sharing statistics between transpositions (identical positions reached by different move orders), or other tree policies, and the evidence for each in the literature (Browne et al. 2012; Silver et al. 2018).

Exercise 8.15 ●●● What makes a learned model useful?

Write a short research essay on this question. Discuss (a) the difference between a model that predicts observations accurately and one that is useful for planning; (b) compounding errors and model exploitation, and how short rollouts (Janner et al. 2019) and planning in a learned latent space (Hafner et al. 2020; Schrittwieser et al. 2020) address them; (c) the argument of Hasselt et al. (2019) about replay as a model; and (d) an experiment, in the maze of this chapter with a function-approximation model of your choice, that would show when planning with the learned model helps and when it hurts.

Part III

Generalization

Real problems have far too many states to tabulate. Part III replaces tables by functions that generalize from the states an agent has seen to the ones it has not. Chapter 9 develops linear methods and their theory, and explains why combining approximation, bootstrapping and off-policy data can make learning diverge. Chapter 10 turns these ideas into deep reinforcement learning, from DQN to distributional methods.

CHAPTER 9

Value Function Approximation

WHAT YOU WILL LEARN

- Explain why large and continuous state spaces call for function approximation, and what the prediction objective $\overline{\text{VE}}$ and its state weighting mean.
- Build features by state aggregation, polynomials, Fourier bases, radial basis functions and tile coding, and choose among them.
- Implement gradient Monte Carlo, semi-gradient TD(0), LSTD and semi-gradient SARSA with linear features.
- Compute the Monte Carlo solution and the TD fixed point of a small problem by hand, and read the projected Bellman equation geometrically.
- Prove that the projected Bellman operator is a contraction on-policy, state the convergence theorem of linear TD, and prove its error bound (●●).
- Explain the deadly triad with Baird's counterexample, and derive the gradient-TD methods that repair it (●●).
- Describe fitted Q iteration, how its errors propagate, and what is known about the limits of learning with function approximation (●●●).

Every method in Part II kept one number for each state, or for each state and action, in a table. A table is easy to update and easy to analyze: changing the entry of one state leaves every other entry alone, and the convergence proofs of Chapter 6 rely on exactly that independence. But a table needs a separate entry for every state, and it learns nothing about a state until it has visited that state. A backgammon position, the joint angles of a robot arm or the pixels of a video game screen come from state spaces so large, or so continuous, that the agent will almost never see the same state twice. A table cannot help it there.

The remedy is to *generalize*: to store a value function with far fewer parameters than there are states, so that what the agent learns about one state also changes its estimates for similar states. This is what supervised learning does when it fits a regression model to examples, and much of this chapter consists of bringing regression into reinforcement learning. The step is less innocent than it looks. In supervised learning the targets are fixed. In reinforcement learning the targets of TD methods are built from the very estimates being learned, the data depend on the policy, and the states we learn about are not always the states we care about. Combining these features with generalization creates new failure modes, and also some of the most interesting theory in the field.

We start with prediction, estimating V^π for a fixed policy, because almost every idea appears there in its clearest form. Section 9.1 states the objective, and Section 9.2 shows how to build

features, using the Trail with two of its states merged as a running example. Section 9.3 turns Monte Carlo and TD learning into gradient-style methods for parameterized values, and Section 9.4 asks where TD converges, which leads to the projected Bellman equation. Section 9.5 (●●) proves that on-policy linear TD converges and bounds its error. Section 9.6 solves for the TD fixed point directly, and Section 9.7 moves to control, with Mountain Car as the example. Section 9.8 shows how function approximation, bootstrapping and off-policy data can together make learning diverge; Section 9.9 (●●) derives the gradient-TD methods that fix this, and explains why minimizing the Bellman residual is not the answer. Section 9.10 introduces fitted Q iteration, the batch method that leads directly to the deep Q-networks of Chapter 10. Section 9.11 (●●●) surveys what theory says about the limits of learning with function approximation. On a first reading, the ● sections 9.1 to 9.4, 9.6 to 9.8 and 9.10 form a complete path.

9.1 Why approximate

Suppose we want to evaluate a policy for the 1000-state random walk of Example 9.2, or for Mountain Car, whose state is a pair of real numbers. A table would need a thousand entries in the first case and infinitely many in the second, and it would learn about each entry only from visits to that exact state. What we want instead is a rule that guesses the value of a state from its description, and that improves as data arrive.

We therefore represent the value function by a parameterized function $V_w(s)$ with a weight vector $w \in \mathbb{R}^d$, where the number of weights d is much smaller than the number of states. A linear function of a few features of the state is the simplest example; a neural network is the most flexible. Changing w to fix the estimate of one state changes the estimates of many others. That is the point, *generalization*, but it is also the difficulty: an update that improves one state can make others worse. In a table, every state can be right at once. With function approximation it is usually impossible to make every estimate exact, so we must decide which errors matter most.

The natural answer is: the errors in states the agent actually visits. We weight the squared error in each state by how often that state occurs under the policy.

Definition 9.1 On-policy distribution

For a policy π in a continuing task, the *on-policy distribution* μ is the stationary distribution of the Markov chain of states under π: $\mu(s') = \sum_s \mu(s) P^\pi(s' \mid s)$. In an episodic task, let $\eta(s)$ be the expected number of visits to the non-terminal state s in one episode (a function of the state, unrelated to the step size η), which satisfies

$$\eta(s') = d_0(s') + \sum_s \eta(s)\, P^\pi(s' \mid s);$$

the on-policy distribution is the fraction of time spent in each state, $\mu(s) = \eta(s) / \sum_{s'} \eta(s')$.

The equation for η says that a visit to s' happens either at the start of an episode or right after a visit to some state s. Here d_0 is the initial-state distribution and $P^\pi(s' \mid s) = \sum_a \pi(a \mid s) P(s' \mid s, a)$

is the state-to-state transition probability under π; transitions into the terminal state are left out, so the rows of P^π over non-terminal states sum to less than one.

Definition 9.2 Mean squared value error

The *mean squared value error* of the weights w is

$$\overline{\mathrm{VE}}(w) = \sum_{s \in \mathcal{S}} \mu(s)\big[V^\pi(s) - V_w(s)\big]^2 = \|V^\pi - V_w\|_\mu^2,$$

where $\|v\|_\mu^2 = \sum_s \mu(s) v(s)^2$ is the squared norm weighted by μ. Its square root, $\sqrt{\overline{\mathrm{VE}}}$, is the *root-mean-squared (RMS) error*.

The Trail as an example. Take the uniform policy on the Trail, $p_0 = p_1 = \frac{1}{2}$, with $\gamma = 0.9$. Every episode visits Start; half of them go on to Middle; a quarter reach Goal. So $\eta = (1, 0.5, 0.25)$, the episode lasts 1.75 steps on average, and

$$\mu(\text{Start}) = \tfrac{4}{7}, \qquad \mu(\text{Middle}) = \tfrac{2}{7}, \qquad \mu(\text{Goal}) = \tfrac{1}{7}.$$

An error in Start therefore counts twice as much as the same error in Middle and four times as much as one in Goal. The true values, from Chapter 3, are $V^\pi = (1.76, 2.8, 4)$.

PITFALL

$\overline{\mathrm{VE}}$ is a means, not an end. In control, what matters is the policy that the values induce, and a value function with a small $\overline{\mathrm{VE}}$ can still rank two actions in the wrong order; a crude one can rank them correctly. Section 9.10 gives an example of each. We use $\overline{\mathrm{VE}}$ because it is the natural yardstick for prediction, not because it is the only one.

9.2 Features

Before choosing an algorithm we must choose a form for V_w. This section is about the most widely studied form, linear functions of features, and about the many ways to construct the features.

9.2.1 Linear value functions

A *feature* is a number $\phi_i(s)$ computed from the state: the position of the car, the square of its velocity, whether the pole leans to the left. A *linear value function* combines d features with weights:

$$V_w(s) = w^\top \phi(s) = \sum_{i=1}^{d} w_i \, \phi_i(s), \qquad \phi(s) = \big(\phi_1(s), \dots, \phi_d(s)\big)^\top.$$

Its gradient with respect to the weights is just the feature vector, $\nabla_w V_w(s) = \phi(s)$, which makes the learning rules of the next section especially simple. For a finite state space it helps to stack the feature vectors as the rows of the $|\mathcal{S}| \times d$ *feature matrix* Φ, so that the vector of all values is

$V_w = \Phi w$. The value functions that can be represented exactly are the linear combinations of the columns of Φ, a d-dimensional subspace of $\mathbb{R}^{|\mathcal{S}|}$.

A table is the special case of *one-hot* features: one feature per state, equal to 1 in that state and 0 elsewhere. Then Φ is the identity matrix, every value function is representable, and each weight is the value of one state. All the interesting cases have far fewer features than states.

Linear methods are worth studying even in the age of deep networks. They are cheap and well understood, they work remarkably well with good features, and the last layer of a deep network is a linear function of the features computed by the layers below it, so everything in this chapter applies to that layer.

9.2.2 State aggregation

The simplest features partition the states into groups and give each group one binary feature: $\phi_i(s) = 1$ if s is in group i and 0 otherwise. This is *state aggregation*. Every state in a group gets the same value, the weight of the group, and an update in any state of the group changes the value of all of them.

Example 9.1 The aggregated Trail

Suppose the agent cannot tell Start from Middle: both get the feature vector $(1, 0)$, while Goal gets $(0, 1)$, and the terminal state End gets $(0, 0)$ because its value is 0 by definition. Then $V_w(\text{Start}) = V_w(\text{Middle}) = w_1$ and $V_w(\text{Goal}) = w_2$. In matrix form, over the three non-terminal states,

$$\Phi = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

No choice of w_1 can be right for both Start (true value 1.76) and Middle (2.8). The best compromise in $\overline{\text{VE}}$ minimizes $\frac{4}{7}(1.76 - w_1)^2 + \frac{2}{7}(2.8 - w_1)^2$, a weighted average:

$$w_1 = \frac{\frac{4}{7} \cdot 1.76 + \frac{2}{7} \cdot 2.8}{\frac{4}{7} + \frac{2}{7}} = \frac{2 \cdot 1.76 + 2.8}{3} \approx 2.107, \qquad w_2 = 4.$$

The approximation overestimates Start by 0.35 and underestimates Middle by 0.69: the larger error goes where it costs less, in the less frequent state. The remaining error is $\overline{\text{VE}} \approx 0.206$, an RMS error of about 0.454. No method using these features can do better in this norm. We will meet this example several times: Section 9.4 shows that TD learning settles somewhere else.

♦

Aggregation is crude, but it is easy to reason about, it never makes estimates explode (we will see why in Section 9.10), and it is the right choice when the states really come in groups that behave alike.

9.2.3 Polynomials, Fourier features and radial basis functions

When the state is a vector of real numbers $s = (s_1, \dots, s_k)$, we can build features as functions of its components. It helps to rescale every component to $[0, 1]$ first, which the library does for all three families below.

Polynomials. The features $\phi_c(s) = s_1^{c_1} s_2^{c_2} \cdots s_k^{c_k}$, for all integer exponent vectors c with entries from 0 to n, give every polynomial of degree at most n in each variable. They interact well with smooth value functions, but high powers of numbers in $[0, 1]$ are nearly indistinguishable, which makes learning ill-conditioned; polynomials are rarely the best choice in practice.

The Fourier basis. Konidaris et al. (2011) proposed the features

$$\phi_c(s) = \cos\bigl(\pi\, c^\top s\bigr), \qquad c \in \{0, 1, \dots, n\}^k,$$

which are the cosine terms of a Fourier series on $[0, 1]^k$. The integer n is the *order*. Low-frequency features capture the overall shape of the value function and high-frequency ones its detail. The Fourier basis is easy to use and often surprisingly good: in the lab, order 5 (six features) approximates the values of the 1000-state random walk with an RMS error of about 0.008, seven times better than ten aggregated groups. A practical recommendation from the same paper is to give each feature its own step size $\eta/\|c\|$ (and η for $c = 0$), so that the high-frequency features take smaller steps.

Radial basis functions (RBFs). An RBF feature responds to how close s is to a center c_i:

$$\phi_i(s) = \exp\Bigl(-\frac{\|s - c_i\|^2}{2\sigma^2}\Bigr).$$

With centers on a grid, the features cover the space with overlapping bumps of width σ, and the value function is a smooth interpolation between them.

All three families share a problem: the number of features grows exponentially with the dimension k. Order n in k dimensions gives $(n + 1)^k$ Fourier features, and a grid of m RBF centers per dimension gives m^k. In more than a handful of dimensions one must keep only some of the features, for example those whose coefficient vector c has at most two non-zero entries.

9.2.4 Tile coding

Tile coding is the feature construction that made linear reinforcement learning practical for problems with a few continuous dimensions. Cover the state space with a grid of cells, called *tiles*; a state activates the one tile it falls in. One grid alone is just state aggregation. Now add more grids, *tilings*, each shifted by a different fraction of a tile (Figure 9.1a). A state activates exactly one tile in every tiling, so with n tilings its feature vector is binary with exactly n ones, and its value is the sum of the n weights of its active tiles.

The overlapping tilings give a resolution finer than one tile, because two states that share a tile in one tiling are usually separated in another. They also give smooth generalization: an update at s moves the weights of its n tiles, so the value of a nearby state changes in proportion to how many of those tiles it shares (Figure 9.1b). The width of a tile sets how far learning spreads; the number

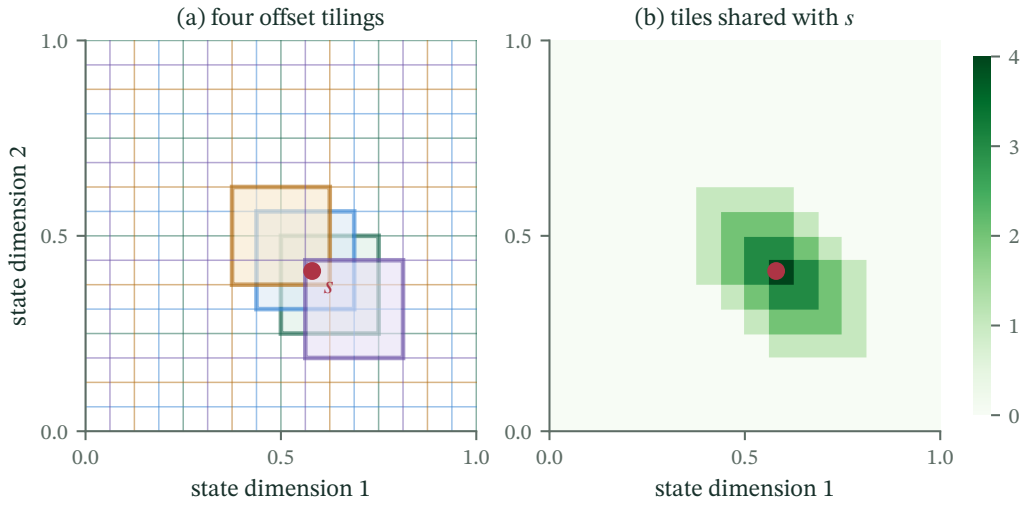


Figure 9.1. Tile coding with four tilings of 4×4 tiles. (a) Each tiling (one color) is shifted by a different fraction of a tile; the state s activates one tile in each, drawn filled. (b) The number of active tiles that each point shares with s. An update at s changes the value of every point by an amount proportional to this number, so it generalizes to a neighborhood of about one tile around s, strongest at s itself.

of tilings sets how finely the function can be shaped. How the tilings are shifted matters too. If every tiling is displaced by the same fraction in every dimension, the generalization pattern is stretched along the diagonal; displacing tiling i by $i \cdot (1, 3, 5, \ldots)$ units of a tile divided by n in the successive dimensions gives a more even pattern (Sutton and Barto 2018, sec. 9.5.4). The library's `TileCoder` uses this displacement.

Tile coding has three practical advantages. Computing a value costs only n additions, whatever the number of tiles. Because exactly n features are active, a natural step size is $\eta = \alpha/n$ for some $\alpha \in (0, 1]$: one update then moves the value of the visited state a fraction α of the way to its target (Exercise 9.4). And the number of tiles can be kept manageable in high dimensions by *hashing* tile coordinates into a fixed-size table, accepting occasional collisions. Tile coding goes back to the CMAC architecture of the 1970s, and Sutton (1996) showed with it that sparse coarse coding solves control problems, such as Mountain Car, on which earlier attempts with function approximation had failed.

9.2.5 Neural networks

All the features so far are designed by hand. A neural network learns its features: a multilayer perceptron computes $V_w(s)$ by passing s through layers of linear maps and nonlinear activations, and all its parameters, the ones that shape the features as well as the final linear layer, are trained together by gradient descent, with gradients computed by backpropagation (Section C.1, Section C.2). Nothing in the learning rules of Section 9.3 requires linearity: they need only the gradient $\nabla_w V_w(s)$. What linearity buys is theory. Most of the guarantees in this chapter hold for linear functions only, and Chapter 10 shows how deep reinforcement learning compensates for their absence with careful engineering.

9.3 Gradient Monte Carlo and semi-gradient TD

With a parameterized value function, learning means adjusting w. If we knew V^π, we could minimize $\overline{\text{VE}}$ by gradient descent: its gradient is

$$\nabla_w \overline{\text{VE}}(w) = -2 \sum_s \mu(s) \big[V^\pi(s) - V_w(s) \big] \nabla_w V_w(s).$$

Following the policy produces states distributed according to μ, so each visited state s_t gives a one-sample estimate of this sum. *Stochastic gradient descent* (SGD) moves the weights a small step against that estimate:

$$w \leftarrow w + \eta \big[U_t - V_w(s_t) \big] \nabla_w V_w(s_t), \tag{9.1}$$

where η is the step size and the factor 2 has been absorbed into it. We have also replaced the unknown $V^\pi(s_t)$ by a *target* U_t that we can observe. In words: move the estimate of the visited state toward the target, and let the gradient decide how that move is shared among the weights. For linear values, $\nabla_w V_w(s_t) = \phi(s_t)$, and the weights of the active features move the most.

9.3.1 Gradient Monte Carlo

The return G_t that follows a visit to s_t is an unbiased sample of $V^\pi(s_t)$: $\mathbb{E}[G_t \mid s_t] = V^\pi(s_t)$. Using it as the target makes (9.1) a genuine stochastic gradient method for $\overline{\text{VE}}$.

Algorithm 9.1 Gradient Monte Carlo prediction

Input: policy π, differentiable V_w, step size $\eta > 0$, discount γ.

1. Initialize w (for example $w = 0$).
2. For each episode:
   1. Generate an episode $s_0, a_0, r_0, \dots, s_{T-1}, a_{T-1}, r_{T-1}, s_T$ by following π.
   2. $G \leftarrow 0$.
   3. For $t = T-1, T-2, \dots, 0$:
      1. $G \leftarrow r_t + \gamma G$.
      2. $w \leftarrow w + \eta \big[G - V_w(s_t) \big] \nabla_w V_w(s_t)$.

Output: $V_w \approx$ the best approximation of V^π in $\overline{\text{VE}}$.

Because the targets are unbiased, the standard theory of stochastic gradient descent applies: with step sizes that decrease according to the Robbins–Monro conditions (Theorem B.1), gradient Monte Carlo converges to a local minimum of $\overline{\text{VE}}$. For linear values $\overline{\text{VE}}$ is a convex quadratic, and the limit is its global minimum, the *Monte Carlo solution*

$$w_{\text{MC}} = \arg\min_w \|V^\pi - \Phi w\|_\mu^2 = (\Phi^\top D \Phi)^{-1} \Phi^\top D V^\pi,$$

where D is the diagonal matrix with the probabilities $\mu(s)$ on its diagonal. On the aggregated Trail this is the weighted average $w_1 \approx 2.107$ of Example 9.1. The price is the price of Monte Carlo in Chapter 5: the agent must wait for the end of the episode, and returns are noisy.

9.3.2 Semi-gradient TD

The TD target of Chapter 6 replaces the rest of the return by the current estimate of the next state's value: $U_t = r_t + \gamma V_w(s_{t+1})$. It is available after one step and has much lower variance than G_t. But it depends on w, and it is biased, because $V_w(s_{t+1})$ is not $V^\pi(s_{t+1})$. If we plug it into (9.1) anyway, we get the most widely used learning rule in reinforcement learning.

Algorithm 9.2 Semi-gradient TD(0)

Input: policy π, differentiable V_w with $V_w(\text{terminal}) = 0$, step size $\eta > 0$, discount γ.

1. Initialize w (for example $w = 0$).
2. For each episode:
   1. Observe the initial state s.
   2. Repeat until the episode ends:
      1. Take $a \sim \pi(\cdot \mid s)$; observe r and s'.
      2. $\delta \leftarrow r + \gamma V_w(s') - V_w(s)$, with $V_w(s') = 0$ if s' is terminal (but not if the episode was only cut off by a time limit).
      3. $w \leftarrow w + \eta\, \delta\, \nabla_w V_w(s)$.
      4. $s \leftarrow s'$.

Output: $V_w \approx$ the TD fixed point (on-policy, linear V_w; see Section 9.4).

The quantity δ is the TD error of (6.1), now computed with the approximate values. With linear values the update is $w \leftarrow w + \eta\, \delta\, \phi(s)$; with one-hot features it reduces to tabular TD(0) (Algorithm 6.1). The distinction between termination and truncation in step 2.2 is the one explained in Section 6.8: a terminal state has value zero, but a state at which a time limit stops the episode still has a value, and the target must bootstrap from it.

Why "semi"? The update looks like a gradient step on the squared TD error $\frac{1}{2}\delta^2$, but it is not one. The target $r + \gamma V_w(s')$ also depends on w, and the true gradient of $\frac{1}{2}\delta^2$ is $-\delta\big[\nabla_w V_w(s) - \gamma \nabla_w V_w(s')\big]$. Semi-gradient TD keeps only the first term: it moves the estimate of s toward the target and deliberately does not move the target. The difference is not small. On the aggregated Trail, a transition from Start to Middle has $\phi(s) = \phi(s') = (1, 0)$, so the full gradient is $-\delta\,(1-\gamma)(1, 0)$: a tenth of the semi-gradient when $\gamma = 0.9$, because changing w_1 moves the estimate and the target almost equally. Keeping the target fixed is what makes TD behave like a prediction method that learns from its own guesses, and Section 9.9.1 shows that the "full gradient" alternative has serious problems of its own. The price is that semi-gradient TD is not gradient descent on any

objective, so the convergence theory of SGD does not apply. We need new tools, and Section 9.5 provides them.

Example 9.2 The 1000-state random walk

The 1000-state random walk of Sutton and Barto (2018, Example 9.1) has states $1, \dots, 1000$, all starting in state 500. Each step jumps to one of the 100 states to the left or one of the 100 states to the right, all equally likely; a jump past an end terminates the episode, with reward -1 on the left and $+1$ on the right, and $\gamma = 1$. Episodes last about 83 steps on average. The true values rise smoothly from about -0.92 near the left end to 0.92 near the right end.

We aggregate the states into ten groups of 100. Figure 9.2a shows the true values together with the two asymptotic solutions, both computed exactly from the transition matrix: the Monte Carlo solution, which is the best staircase in $\overline{\text{VE}}$, and the TD fixed point of the next section. The Monte Carlo staircase follows the true values closely (RMS error 0.054); the TD staircase is noticeably flatter (RMS error 0.117). Figure 9.2b shows learning curves. Semi-gradient TD with $\eta = 0.02$ reaches its asymptote within about 400 episodes. Gradient Monte Carlo needs a step size ten to a hundred times smaller to average out the noise of the returns; with $\eta = 2 \times 10^{-4}$ it is still approaching its better asymptote after 3000 episodes, and with $\eta = 10^{-3}$ its noise keeps it near 0.1.

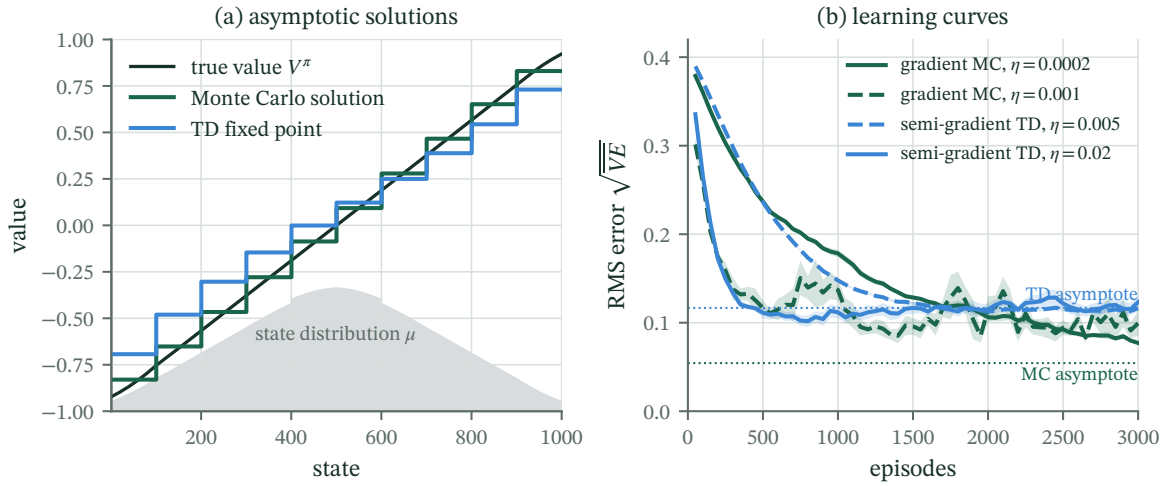


Figure 9.2. The 1000-state random walk with ten aggregated groups. (a) True values, the Monte Carlo solution (the best approximation in the on-policy norm) and the TD fixed point, computed exactly; the gray hump is the on-policy distribution μ (with a spike at the start state 500, not drawn to scale). (b) RMS error of gradient Monte Carlo and semi-gradient TD(0) during learning, mean of 8 runs with a band of ± 1 standard error; dotted lines mark the two asymptotes. TD learns fast but stops at a worse solution; Monte Carlo is slow and noisy but heads for the better one.

This example contains the central trade-off of the chapter. Bootstrapping speeds learning and lowers variance, but with function approximation it changes the answer, not only the speed. We now find out exactly where TD converges.

LAB

Lab 9.1: Value function approximation (`labs/ch09_vfa.py`, about 60 minutes). Run `python3 labs/ch09_vfa.py`. Task 1 learns the aggregated Trail with gradient Monte Carlo and semi-gradient TD and compares them with the exact solutions; Task 2 repeats the comparison on the random walk; Task 3 measures how many Fourier features the random walk needs; later tasks cover LSTD, Mountain Car, Baird's counterexample and fitted Q iteration.

9.4 The TD fixed point

Tabular TD converges to V^π. Linear TD cannot, in general, because V^π is usually not representable. Where does it go?

The expected update. Take linear values and suppose states are visited according to the on-policy distribution μ. Write $r^\pi(s)$ for the expected reward in s and P^π for the transition matrix under π, as in (3.9). Averaging the update $\delta_t \phi(s_t)$ first over the transition from $s_t = s$ and then over $s \sim \mu$ gives

$$\mathbb{E}\big[\delta_t\, \phi(s_t)\big] = \sum_s \mu(s)\, \phi(s) \Big[r^\pi(s) + \gamma \sum_{s'} P^\pi(s' \mid s)\, \phi(s')^\top w - \phi(s)^\top w \Big] = b - Aw,$$

with the $d \times d$ matrix A and the vector b

$$A = \Phi^\top D\,(I - \gamma P^\pi)\,\Phi, \qquad b = \Phi^\top D\, r^\pi. \tag{9.2}$$

Semi-gradient TD therefore follows, on average, the linear dynamics $w \leftarrow w + \eta\,(b - Aw)$. If it converges, the expected update must vanish at the limit, and the limit is the solution of the linear system $Aw = b$:

$$w_{\mathrm{TD}} = A^{-1} b.$$

This is the *TD fixed point*. Section 9.5 proves that on-policy linear TD does converge to it, and that A is invertible when the features are linearly independent.

The projected Bellman equation. The TD fixed point has a beautiful geometric meaning. Let Π be the matrix that maps a value function v to its best approximation in the μ-weighted norm,

$$\Pi v = \Phi\,(\Phi^\top D \Phi)^{-1} \Phi^\top D\, v, \qquad \text{so that } \Pi v = \Phi w \text{ with } w = \arg\min_w \|v - \Phi w\|_\mu.$$

Π is the orthogonal projection onto the representable functions, orthogonal in the inner product $\langle u, v \rangle_\mu = \sum_s \mu(s) u(s) v(s)$; the Monte Carlo solution is $\Phi w_{\mathrm{MC}} = \Pi V^\pi$. Let $\mathcal{T}^\pi v = r^\pi + \gamma P^\pi v$ be the Bellman operator of π (Section 4.3). The condition $Aw = b$ reads $\Phi^\top D\,(\mathcal{T}^\pi \Phi w - \Phi w) = 0$: the Bellman error $\mathcal{T}^\pi \Phi w - \Phi w$ is orthogonal to every feature. Multiplying by $\Phi(\Phi^\top D \Phi)^{-1}$ turns this into $\Pi(\mathcal{T}^\pi \Phi w - \Phi w) = 0$, and since $\Pi \Phi w = \Phi w$,

$$\Phi w_{\mathrm{TD}} = \Pi\, \mathcal{T}^\pi\, \Phi w_{\mathrm{TD}}. \tag{9.3}$$

This is the *projected Bellman equation*. In words: the TD fixed point is the representable value function that reproduces itself when we apply one Bellman backup and project the result back onto the representable functions. The tabular Bellman equation $V = \mathcal{T}^\pi V$ is the case $\Pi = I$.

Example 9.3 The TD fixed point of the aggregated Trail

For the aggregated Trail under the uniform policy, the chain over (Start, Middle, Goal) has

$$P^\pi = \begin{pmatrix} 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \\ 0 & 0 & 0 \end{pmatrix}, \qquad r^\pi = \begin{pmatrix} 0.5 \\ 1 \\ 4 \end{pmatrix}, \qquad D = \tfrac{1}{7}\,\mathrm{diag}(4, 2, 1).$$

(The exits and the finish lead to the terminal state, which contributes nothing.) Then

$$A = \frac{1}{7}\begin{pmatrix} 4.2 & -0.9 \\ 0 & 1 \end{pmatrix}, \qquad b = \frac{1}{7}\begin{pmatrix} 4 \\ 4 \end{pmatrix}.$$

The second equation gives $w_2 = 4$, and the first $4.2\,w_1 = 4 + 0.9 \cdot 4 = 7.6$, so

$$w_{\mathrm{TD}} = (1.810,\ 4).$$

Compare the Monte Carlo solution $(2.107, 4)$: TD values the merged state 0.3 lower. The reason is bootstrapping. From Start, the action go leads to Middle, and TD's target uses the estimate of Middle, which is w_1 itself, not the true 2.8. TD therefore believes that going on from Start is worth 0.9×1.81, not 0.9×2.8, and a low estimate feeds a low target. Monte Carlo sees actual returns and is not fooled. The cost is visible in the error: $\overline{\mathrm{VE}}(w_{\mathrm{TD}}) \approx 0.282$, an RMS error of 0.531, against 0.454 for Monte Carlo.

The projection here is easy to write down: Π replaces the values of Start and Middle by their average with weights $\frac{2}{3}$ and $\frac{1}{3}$ and leaves Goal alone. Applying $\mathcal{T}^\pi$ to $\Phi w_{\mathrm{TD}} = (1.81, 1.81, 4)$ gives $(0.5 + 0.45 \cdot 1.81,\ 1 + 0.45 \cdot 4,\ 4) = (1.31, 2.8, 4)$, and projecting gives back $\frac{2}{3} \cdot 1.31 + \frac{1}{3} \cdot 2.8 = 1.81$ in the first two coordinates, as (9.3) says. Iterating $v \leftarrow \Pi\mathcal{T}^\pi v$ from $v = 0$ gives $w_1 = 0.67, 1.47, 1.71, 1.78, 1.80, \ldots$, converging to 1.81; once $w_2 = 4$, each iteration maps w_1 to $1.267 + 0.3\,w_1$.

♦

Figure 9.3 draws this example. Because Goal is represented exactly, every point of interest has $V(\mathrm{Goal}) = 4$, and the geometry fits in the plane of the other two coordinates. Scaling the axes by $\sqrt{\mu}$ turns the μ-weighted projection into an ordinary perpendicular one.

Which is better? Monte Carlo finds the best approximation, so in $\overline{\mathrm{VE}}$ the TD fixed point can only be worse. How much worse? Theorem 9.5 answers: never more than a factor $1/\sqrt{1-\gamma^2}$ in RMS error, on-policy. With $\gamma = 0.9$ that factor is 2.29, and on the aggregated Trail the actual ratio is $0.531/0.454 = 1.17$. The bound grows as $\gamma \to 1$, and on the random walk, with $\gamma = 1$, TD's error is more than twice Monte Carlo's. Against this, TD learns online, has lower variance, and is usually much faster, as Figure 9.2b showed. In practice the speed usually wins, and multi-step returns and eligibility traces (Chapter 7) let one interpolate: with λ-returns the fixed point moves toward the Monte Carlo solution as $\lambda \to 1$ (Tsitsiklis and Van Roy 1997).

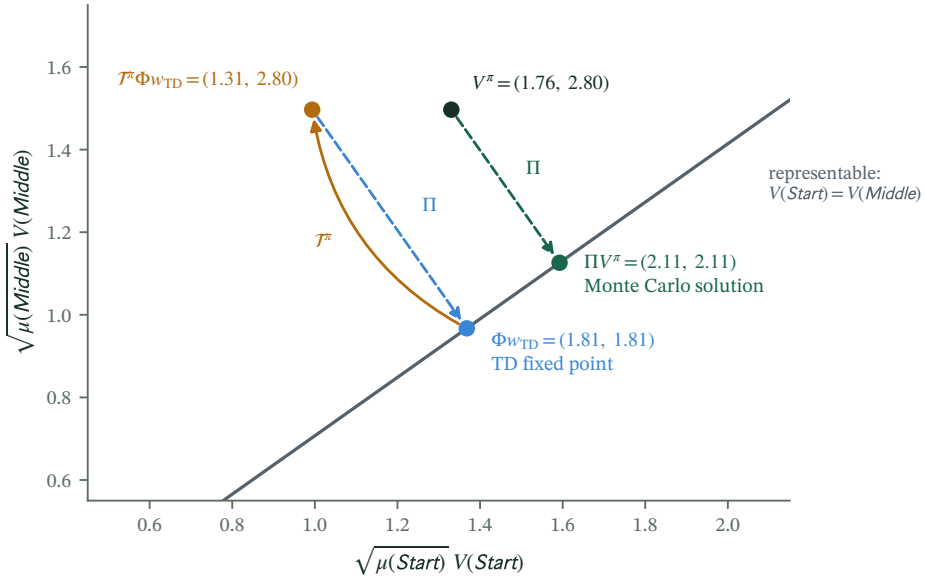


Figure 9.3. The geometry of the aggregated Trail. The line holds the representable value functions, $V(\text{Start}) = V(\text{Middle})$; axes are scaled by $\sqrt{\mu}$, so Π is a perpendicular projection. Monte Carlo finds ΠV^π, the point of the line closest to the true value. TD finds the point Φw_{TD} whose Bellman backup $\mathcal{T}^\pi \Phi w_{\text{TD}}$ projects back onto itself. The two answers differ, and TD's is farther from V^π. Every point drawn has $V(\text{Goal}) = 4$.

CHECKPOINT

1. On the aggregated Trail, why does TD learn $V_w(\text{Goal}) = 4$ exactly, while its value for the merged state is biased?
2. If Φ is the identity matrix (a table), what are Π and the TD fixed point?
3. Is the TD fixed point the minimizer of some error between Φw and V^π?

Answers.

1. Goal has its own feature and its target, the reward 4, does not bootstrap from any approximate value. The merged state's target bootstraps from w_1 itself.
2. $\Pi = I$, so the projected Bellman equation is the Bellman equation and $\Phi w_{\text{TD}} = V^\pi$.
3. Not of $\|\Phi w - V^\pi\|_\mu$: that is minimized by Monte Carlo. It is the zero of the projected Bellman error $\|\Pi(\mathcal{T}^\pi \Phi w - \Phi w)\|_\mu$ (Section 9.9).

9.5 Convergence and error of linear TD ●●

This section proves the facts used above: on-policy, the projected Bellman operator is a contraction, linear TD converges to its fixed point, and the fixed point is not much worse than the best approximation. Throughout, the task is continuing, with a finite state space, a policy whose state chain is irreducible and aperiodic with stationary distribution $\mu > 0$, $\gamma < 1$, and features whose columns are linearly independent (so $\Phi^\top D\Phi$ is invertible). We return to episodic tasks at the end.

Lemma 9.3 $\Pi\mathcal{T}^\pi$ is a γ-contraction in $\|\cdot\|_\mu$

If μ is the stationary distribution of P^π, then $\|P^\pi v\|_\mu \le \|v\|_\mu$ for every v, and

$$\|\Pi\mathcal{T}^\pi u - \Pi\mathcal{T}^\pi v\|_\mu \le \gamma\,\|u - v\|_\mu \quad \text{for all } u, v.$$

Proof. *Step 1: P^π does not expand $\|\cdot\|_\mu$.* By Jensen's inequality, $(\sum_{s'} P^\pi(s' \mid s)v(s'))^2 \le \sum_{s'} P^\pi(s' \mid s)v(s')^2$, because each row of P^π is a probability distribution. Hence

$$\|P^\pi v\|_\mu^2 = \sum_s \mu(s)\Big(\sum_{s'} P^\pi(s' \mid s)v(s')\Big)^2 \le \sum_{s'} \underbrace{\Big(\sum_s \mu(s)P^\pi(s' \mid s)\Big)}_{=\mu(s') \text{ by stationarity}} v(s')^2 = \|v\|_\mu^2.$$

Step 2: Π does not expand $\|\cdot\|_\mu$. Π is the orthogonal projection in the inner product $\langle\cdot,\cdot\rangle_\mu$: it is linear, $\Pi^2 = \Pi$, and $v - \Pi v$ is orthogonal to every representable function, in particular to Πv. By Pythagoras, $\|v\|_\mu^2 = \|\Pi v\|_\mu^2 + \|v - \Pi v\|_\mu^2 \ge \|\Pi v\|_\mu^2$.

Step 3. $\Pi\mathcal{T}^\pi u - \Pi\mathcal{T}^\pi v = \gamma\,\Pi P^\pi(u - v)$, because the rewards cancel. By steps 2 and 1, its norm is at most $\gamma\|P^\pi(u - v)\|_\mu \le \gamma\|u - v\|_\mu$. ■

By Banach's fixed-point theorem (Theorem 4.4) applied on the representable subspace, $\Pi\mathcal{T}^\pi$ has a unique fixed point there, which is Φw_{TD}; and since the columns of Φ are independent, w_{TD} is unique. The same computation shows that A is *positive definite* in the sense that $x^\top Ax > 0$ for every $x \ne 0$ (although A is not symmetric). With $v = \Phi x \ne 0$,

$$x^\top Ax = \|v\|_\mu^2 - \gamma\langle v, P^\pi v\rangle_\mu \ge \|v\|_\mu^2 - \gamma\|v\|_\mu\|P^\pi v\|_\mu \ge (1 - \gamma)\|v\|_\mu^2 > 0,$$

by the Cauchy–Schwarz inequality and step 1. In particular A is invertible, and every eigenvalue of A has a positive real part.

Theorem 9.4 Convergence of on-policy linear TD (Tsitsiklis and Van Roy 1997)

Under the assumptions above, suppose the states s_t are generated by following π (so they form a Markov chain with stationary distribution μ), the rewards are bounded, and the step sizes $\eta_t > 0$ are deterministic, nonincreasing, and satisfy $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$. Then the weights of linear semi-gradient TD(0), $w_{t+1} = w_t + \eta_t\,\delta_t\,\phi(s_t)$, converge with probability one to the TD fixed point $w_{\mathrm{TD}} = A^{-1}b$.

Proof. *Sketch.* The update is $w_{t+1} = w_t + \eta_t\,(b_t - A_t w_t)$ with $A_t = \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^\top$ and $b_t = r_t\phi(s_t)$, whose averages over the stationary chain are A and b. The ODE method of stochastic approximation (Theorem B.4; Borkar (2008)) says that, with Robbins–Monro step sizes and suitably controlled noise, the iterates track the solutions of the ordinary differential equation $\dot w = b - Aw$. This ODE has the single equilibrium w_{TD}, and it is globally asymptotically stable: the function $L(w) = \|w - w_{\mathrm{TD}}\|^2$ decreases along solutions, since $\dot L = -2(w - w_{\mathrm{TD}})^\top A\,(w - w_{\mathrm{TD}}) < 0$ for $w \ne w_{\mathrm{TD}}$ by the positive definiteness of A. The remaining work, which is the technical heart of Tsitsiklis and Van Roy (1997), is to control the noise, which is not a martingale difference because consecutive states are dependent; they do so with results on stochastic approximation driven by ergodic Markov chains. Their theorem covers TD(λ) for every $\lambda \in [0, 1]$. ■

Theorem 9.5 Error of the TD fixed point

Under the assumptions above,

$$\|\Phi w_{\mathrm{TD}} - V^\pi\|_\mu \le \frac{1}{\sqrt{1-\gamma^2}} \|\Pi V^\pi - V^\pi\|_\mu.$$

Proof. The error splits into a part orthogonal to the representable functions and a part inside them: $\Phi w_{\mathrm{TD}} - V^\pi = (\Phi w_{\mathrm{TD}} - \Pi V^\pi) + (\Pi V^\pi - V^\pi)$. The first part is representable and the second is orthogonal to every representable function (it is the residual of a projection), so by Pythagoras

$$\|\Phi w_{\mathrm{TD}} - V^\pi\|_\mu^2 = \|\Phi w_{\mathrm{TD}} - \Pi V^\pi\|_\mu^2 + \|\Pi V^\pi - V^\pi\|_\mu^2.$$

For the first term, use $\Phi w_{\mathrm{TD}} = \Pi \mathcal{T}^\pi \Phi w_{\mathrm{TD}}$, $V^\pi = \mathcal{T}^\pi V^\pi$ and the lemma:

$$\|\Phi w_{\mathrm{TD}} - \Pi V^\pi\|_\mu = \|\Pi \mathcal{T}^\pi \Phi w_{\mathrm{TD}} - \Pi \mathcal{T}^\pi V^\pi\|_\mu \le \gamma \|\Phi w_{\mathrm{TD}} - V^\pi\|_\mu.$$

Substituting, $\|\Phi w_{\mathrm{TD}} - V^\pi\|_\mu^2 \le \gamma^2 \|\Phi w_{\mathrm{TD}} - V^\pi\|_\mu^2 + \|\Pi V^\pi - V^\pi\|_\mu^2$; rearranging and taking square roots gives the bound. ■

In words: the TD fixed point is within a constant factor of the best possible error, and the constant depends only on the discount. If the features can represent V^π exactly, TD finds it. Tsitsiklis and Van Roy (1997) state the bound with the factor $(1-\lambda\gamma)/(1-\gamma)$ for TD(λ), which for $\lambda = 0$ is the weaker $1/(1-\gamma)$ (Exercise 9.11); the Pythagorean argument above, common in later treatments, is sharper. For the aggregated Trail the bound is $2.29 \times 0.454 = 1.04$, and the true error is 0.531.

Episodic tasks. The Trail and the random walk are episodic, and their chains have no stationary distribution over non-terminal states. The proofs above use stationarity only through step 1 of the lemma, and for that the inequality $\sum_s \mu(s) P^\pi(s' \mid s) \le \mu(s')$ is enough. The episodic on-policy distribution of Definition 9.1 satisfies it, because $\sum_s \eta(s) P^\pi(s' \mid s) = \eta(s') - d_0(s') \le \eta(s')$ (Exercise 9.10). So the lemma, the positive definiteness of A and the error bound hold there too, for $\gamma < 1$; this is why our Trail numbers obey the theory.

Where the proof breaks off-policy. Every step that used μ required it to be the on-policy distribution. If the states are sampled from some other distribution, for instance because the data come from a different behavior policy, $\|P^\pi v\|$ can exceed $\|v\|$ in the sampling norm, $\Pi\mathcal{T}^\pi$ can be an expansion, and A can have eigenvalues with negative real parts. Then the expected update $w \leftarrow w + \eta(b - Aw)$ pushes the weights away from the fixed point, for every step size. Section 9.8 shows this happening.

RESEARCH NOTE

The theorem above is asymptotic. Finite-time analyses bound the mean squared distance of the iterates from w_{TD} after t steps, with constants that depend on the smallest eigenvalue of the symmetric part of A and on how fast the chain mixes (Bhandari et al. 2018; Srikant and Ying 2019). How to choose step sizes and features so that these constants are favorable in large problems is still poorly understood.

9.6 Least-squares TD

The TD fixed point is the solution of a linear system, $Aw = b$. Semi-gradient TD approaches it by many small, noisy steps whose size we must tune. Why not estimate A and b from the data and solve the system directly? That is *least-squares TD* (LSTD) (Bradtke and Barto 1996; Boyan 2002). After t transitions,

$$\hat{A}_t = \varepsilon I + \sum_{k=0}^{t-1} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^\top, \qquad \hat{b}_t = \sum_{k=0}^{t-1} r_k\,\phi(s_k), \qquad w_t = \hat{A}_t^{-1}\hat{b}_t,$$

with $\phi(s_{k+1}) = 0$ when s_{k+1} is terminal. Without the regularizer, dividing both sums by t would not change w_t. The small multiple εI of the identity makes $\hat{A}_t$ invertible from the start, and its influence fades as the sums grow. As $t \to \infty$, $\hat{A}_t/t \to A$ and $\hat{b}_t/t \to b$ for on-policy data, so $w_t \to w_{\mathrm{TD}}$.

Solving the system from scratch after every step would cost $O(d^3)$. Instead, we can keep the inverse $C = \hat{A}^{-1}$ and update it with the Sherman–Morrison formula for the inverse of a matrix plus an outer product (Exercise 9.12):

Algorithm 9.3 LSTD with incremental inverse

Input: policy π, features ϕ with $\phi(\text{terminal}) = 0$, discount γ, small $\varepsilon > 0$.

1. $C \leftarrow \varepsilon^{-1} I$ (the inverse of $\hat{A} = \varepsilon I$); $\hat{b} \leftarrow 0$.
2. For each transition (s, r, s') generated by following π:
   1. $x \leftarrow \phi(s)$, $x' \leftarrow \phi(s')$.
   2. $v \leftarrow C^\top(x - \gamma x')$.
   3. $C \leftarrow C - \dfrac{(Cx)\,v^\top}{1 + v^\top x}$.
   4. $\hat{b} \leftarrow \hat{b} + r\,x$.
   5. $w \leftarrow C\,\hat{b}$.

Output: $w \approx w_{\mathrm{TD}}$.

Each step costs $O(d^2)$ operations and $O(d^2)$ memory, against $O(d)$ for semi-gradient TD. In return, LSTD has no step size and extracts much more from each sample. For example, with order-5 Fourier features on the random walk and batches of 300 episodes (about 25,000 transitions), LSTD's RMS error averaged 0.046 over eight batches. One pass of semi-gradient TD over the same transitions averaged 0.069 with the best step size we tried ($\eta = 0.01$, scaled per feature), and did markedly worse with a step size three times larger or smaller. LSTD has no step size to get wrong. With d in the thousands, as with tile coding, the quadratic cost becomes a burden, and semi-gradient TD is usually preferred.

LSTD has two further limitations. It weights all past data equally, so it cannot forget; if the policy changes, as in control, the old data describe the wrong policy. And the regularizer ε must

be chosen with some care: too large, and it biases w toward zero; too small, and early estimates are erratic.

LSPI. *Least-squares policy iteration* (Lagoudakis and Parr 2003) solves the first problem by keeping a fixed batch of transitions (s, a, r, s') and alternating two steps, as policy iteration does (Algorithm 4.2). For evaluation, it computes the TD fixed point of Q^π for features $\phi(s, a)$ of state-action pairs, using as next action the action of the current policy in s' (this is *LSTDQ*). For improvement, it makes the policy greedy with respect to the new Q. Because the next action is chosen by the policy being evaluated, the same data can be reused for every policy. LSPI often converges in a handful of iterations; the library function `lspi` implements it, and Lazaric et al. (2012) give finite-sample guarantees.

9.7 Control with function approximation

For control we approximate action values, $Q_w(s, a) \approx Q^\pi(s, a)$, and improve the policy by acting greedily, or ε-greedily, with respect to them: generalized policy iteration (Section 4.6) with approximate evaluation. The features now describe a state and an action. A simple and common choice keeps one weight vector per action, $Q_w(s, a) = w_a^\top \phi(s)$; this is what we do with tile coding. Replacing V_w by Q_w in semi-gradient TD gives the control version of SARSA (Algorithm 6.2).

Algorithm 9.4 Episodic semi-gradient SARSA

Input: differentiable Q_w with $Q_w(\text{terminal}, \cdot) = 0$, step size $\eta > 0$, exploration rate $\varepsilon \geq 0$, discount γ.

1. Initialize w (for example $w = 0$).
2. For each episode:
   1. Observe s; choose a ε-greedily with respect to $Q_w(s, \cdot)$.
   2. Repeat until the episode ends:
      1. Take a; observe r and s'.
      2. If s' is terminal: $w \leftarrow w + \eta \big[r - Q_w(s, a)\big] \nabla_w Q_w(s, a)$, and end the episode.
      3. Choose a' ε-greedily with respect to $Q_w(s', \cdot)$.
      4. $w \leftarrow w + \eta \big[r + \gamma Q_w(s', a') - Q_w(s, a)\big] \nabla_w Q_w(s, a)$.
      5. $s \leftarrow s'$, $a \leftarrow a'$.

Output: Q_w and its ε-greedy policy.

Using $\max_{a'} Q_w(s', a')$ in step 2.4 instead gives semi-gradient Q-learning. That target evaluates the greedy policy from data generated by an exploratory one, so it is off-policy, and Section 9.8 shows why that matters.

Example 9.4 Mountain Car

An underpowered car sits in a valley and must reach the top of the hill on the right (Moore 1990; Sutton and Barto 2018, Example 10.1). The state is the position $x \in [-1.2, 0.5]$ and the velocity $v \in [-0.07, 0.07]$. The actions are full reverse, coast and full forward, $a \in \{-1, 0, 1\}$, and the dynamics are $v \leftarrow \text{clip}(v + 0.001\,a - 0.0025\cos 3x, -0.07, 0.07)$, then $x \leftarrow x + v$. Every step earns the reward -1 until x reaches 0.5, so the return is minus the number of steps, and $\gamma = 1$. Gravity is stronger than the engine: from the bottom of the valley, full throttle forward stalls on the slope. The car must first back up the left slope and then use the momentum. This is the difficulty: to get to the goal, the agent must first move away from it.

We use tile coding with 8 tilings of 8×8 tiles over the box of positions and velocities, which with the extra row and column for the offsets is $8 \times 9 \times 9 = 648$ features per action, and semi-gradient SARSA with $\eta = 0.5/8$ and $\varepsilon = 0$. Exploration comes from the initial weights: $Q_w = 0$ is optimistic when every reward is -1, so untried actions look better than tried ones. The first episode takes well over a thousand steps (about 1400 on average). After a hundred episodes the car reaches the goal in about 140 steps. Figure 9.4 shows the learned cost to go, $-\max_a Q_w(s, a)$, and learning curves for three step sizes.

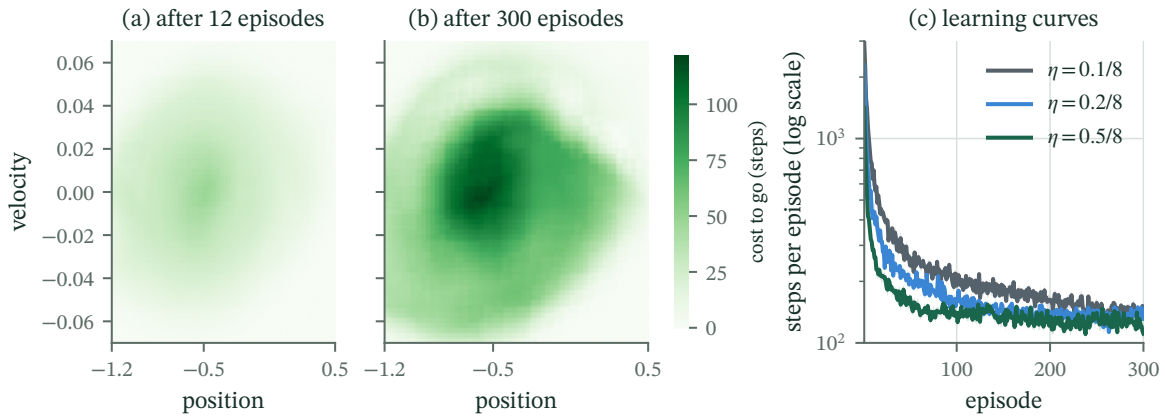


Figure 9.4. Semi-gradient SARSA with tile coding on Mountain Car. (a, b) The learned number of steps to the goal from each state, $-\max_a Q_w(s, a)$, after 12 and after 300 episodes; darker is farther. The spiral is the structure of the problem: from the valley floor the car must swing back and forth, and each extra swing adds a ring of cost. (c) Steps per episode (log scale) for three step sizes, mean of 10 runs with ±1 standard error bands. All three learn; larger steps learn faster here.

The theory of control with function approximation is much thinner than that of prediction. With linear features, semi-gradient SARSA's weights stay bounded, but they need not converge: as the greedy policy switches between actions, the weights can chatter within a region (Gordon 2001). Convergence can be proved when the policy changes smoothly enough with the values (Perkins and Precup 2003). In practice, SARSA with tile coding is reliable on problems of this size, which is why it was the workhorse of reinforcement learning before deep networks.

9.7.1 Continuing tasks and average reward ●●

In a continuing task there are no episodes, and with function approximation discounting raises a conceptual problem. In the tabular case a single policy is optimal in every state at once. With function approximation it usually is not, so we must rank policies by a single number, and the natural one is the average of the values over the states the policy visits. But if μ_π is the stationary distribution of π,

$$\sum_s \mu_\pi(s)\, V_\gamma^\pi(s) = \frac{r(\pi)}{1-\gamma}, \qquad r(\pi) = \sum_s \mu_\pi(s) \sum_a \pi(a \mid s)\, r(s,a),$$

for every $\gamma < 1$ (Exercise 9.15). The ranking of policies by this criterion does not depend on γ at all: it is the ranking by the *average reward* $r(\pi)$ of Section 3.10.2. This is the argument of Sutton and Barto (2018, sec. 10.4) for treating continuing control problems with function approximation in the average-reward setting.

Average-reward methods learn *differential* values, which measure how much better than average the future is from a state, together with an estimate $\bar{R}$ of $r(\pi)$. Differential semi-gradient SARSA uses the TD error

$$\delta = r - \bar{R} + Q_w(s', a') - Q_w(s, a)$$

and updates $\bar{R} \leftarrow \bar{R} + \beta\,\delta$ and $w \leftarrow w + \eta\,\delta\,\nabla_w Q_w(s,a)$, with a second step size β; there is no discount. Its convergence theory, like that of discounted SARSA, is incomplete.

9.8 The deadly triad

We have seen on-policy linear TD converge with a guarantee. Now we see what can go wrong. Sutton and Barto (2018, sec. 11.3) call it the *deadly triad*: instability and divergence can occur when three ingredients are combined,

- *function approximation*, which makes an update in one state change the values of others;
- *bootstrapping*, whose targets are built from the current estimates; and
- *off-policy training*, in which the distribution of the updates is not the on-policy distribution of the policy whose values appear in the targets.

Any two of the three are safe in the settings we have analyzed. Without bootstrapping, gradient Monte Carlo is stochastic gradient descent on a fixed objective. Without off-policy training, Theorem 9.4 applies. Without function approximation, tabular methods converge, as Chapter 6 showed for Q-learning. All three together have no such protection.

The simplest example. Take two states whose features are the numbers 1 and 2, with a single weight, so their estimates are w and $2w$ (Sutton and Barto 2018, sec. 11.2). A transition from the first state to the second earns reward 0. Its TD error is $\delta = 0 + \gamma \cdot 2w - w = (2\gamma - 1)\,w$, and the update is

$$w \leftarrow w + \eta\,(2\gamma - 1)\,w \cdot 1 = \big(1 + \eta(2\gamma - 1)\big)\,w.$$

For $\gamma > \frac{1}{2}$ this multiplies w by a number larger than one. If the data contain only this transition, as can happen off-policy, w grows geometrically from any nonzero start, though the true values

(both 0 if the second state ends the episode with reward 0) are representable with $w = 0$. With $\eta = 0.1$ and $\gamma = 0.9$, each update multiplies w by 1.08. The three legs are visible. Raising w to fit the first state raises the second state's estimate twice as much (function approximation); the first state's target is built from that estimate (bootstrapping); and the second state's own transition, which would pull w back down, is never updated (off-policy). On-policy, the second state is updated too: if its transition ends the episode with reward 0, its update multiplies w by $1 - 4\eta$, and one episode multiplies it by $(1.08)(0.6) = 0.648$ when $\eta = 0.1$. The estimates decay to the truth (Exercise 9.6).

The example is small but artificial: why would off-policy data contain only one of the two transitions? Baird's counterexample shows that divergence happens even when every state is visited.

Example 9.5 Baird's counterexample

The Markov decision process of Baird (1995) has seven states and two actions ((Sutton and Barto 2018, Example 11.1)). The *dashed* action moves to one of the six "upper" states, chosen uniformly; the *solid* action moves to the seventh, "lower" state. Every reward is 0 and $\gamma = 0.99$, so every true value is 0. The values are linear in eight weights:

$$V_w(i) = 2w_i + w_8 \quad (i = 1, \dots, 6), \qquad V_w(7) = w_7 + 2w_8.$$

The true values are representable, by $w = 0$. The *target policy* π always takes the solid action; the *behavior policy* b takes the dashed action with probability $\frac{6}{7}$ and the solid one with probability $\frac{1}{7}$, so that the next state is uniformly distributed over all seven states. Off-policy TD(0) corrects for the difference between the policies with the importance-sampling ratio $\rho_t = \pi(a_t \mid s_t)/b(a_t \mid s_t)$ of Section 5.4, which is 7 for the solid action and 0 for the dashed one:

$$w \leftarrow w + \eta\, \rho_t\, \delta_t\, \phi(s_t).$$

Starting from $w = (1, 1, 1, 1, 1, 1, 10, 1)$ with $\eta = 0.01$, the weights grow without bound (Figure 9.5a): after 1000 steps the norm of w is about 390.

The growth is not noise. In expectation the update is $w \leftarrow w + \eta\,(b - Aw)$ with $A = \Phi^\top D\,(I - \gamma P^\pi)\Phi$, where now D holds the behavior's uniform state distribution and P^π sends every state to the lower state. This A has two eigenvalues with negative real parts, about -0.24 and -0.02. Along the corresponding directions the expected update multiplies the weights by $1 + 0.24\,\eta > 1$, for every step size $\eta > 0$. Even exact, noise-free synchronous updates of all states (dynamic programming with linear approximation) diverge.

♦

Notice what the importance-sampling ratio does and does not do. It corrects the distribution of *actions* in each state, so that each update has the right expected target. It does not correct the distribution of *states*: the updates still arrive with the behavior policy's state frequencies, and it is the mismatch between those frequencies and the target policy's dynamics that breaks the contraction of Lemma 9.3. Under the target policy the chain spends all its time in the lower state, and an update distribution concentrated there would be stable.

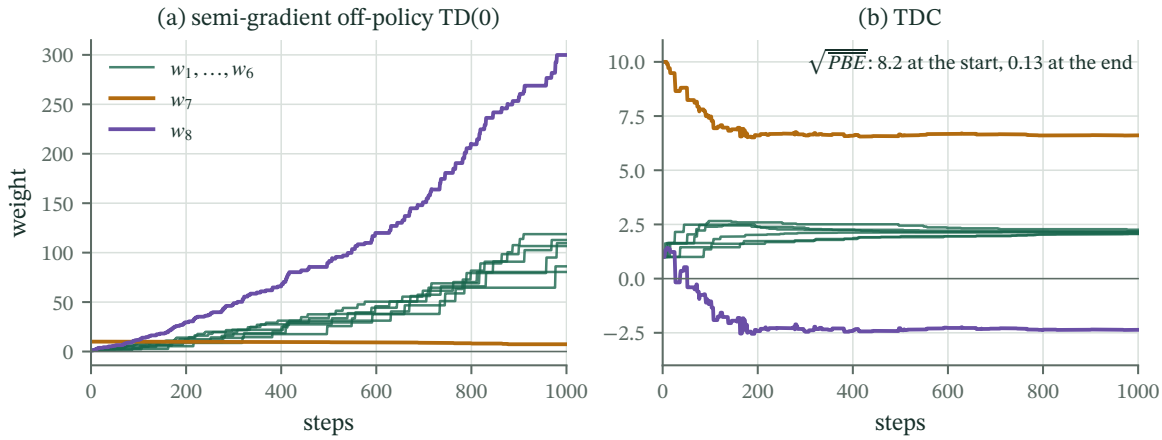


Figure 9.5. Baird's counterexample, starting from $w = (1, 1, 1, 1, 1, 1, 10, 1)$. (a) Semi-gradient off-policy TD(0) with $\eta = 0.01$: the weights grow without bound, although $w = 0$ represents the true values exactly. (b) TDC (Algorithm 9.5) with $\eta = 0.005$ and $\eta_u = 0.05$ on the same stream of experience: the weights stay bounded and settle, and the projected Bellman error falls by a factor of about sixty.

PITFALL

The deadly triad is a risk, not a verdict, and also not the only risk. Many off-policy methods with bootstrapping and function approximation work well in practice, including the deep Q-networks of Chapter 10, which combine all three legs. Conversely, nonlinear function approximation can diverge even on-policy (Tsitsiklis and Van Roy 1997). What the triad tells you is where the guarantees end, and which change (fewer bootstrapping steps, data closer to on-policy, a different algorithm) is likely to restore stability when values blow up.

How can we keep all three ingredients and still have a guarantee? Three answers have been developed. Gradient-TD methods (Section 9.9) do true stochastic gradient descent on an objective whose minimum is the TD fixed point. Emphatic methods (Section 9.9.2) reweight the updates so that the contraction is restored. And in deep reinforcement learning, target networks and replay buffers (Chapter 10) slow down the feedback loop without removing it; Hasselt et al. (2018) study how far that goes.

9.9 Gradient-TD methods ●●

Semi-gradient TD is not the gradient of anything, which is why the step size cannot save it in Baird's example. To get a guarantee we need an objective and its true gradient. Four candidate objectives are worth distinguishing. Write $\delta_w = \mathcal{T}^\pi \Phi w - \Phi w$ for the vector of expected TD errors, the *Bellman error* in each state:

Table 9.1. Four objectives for value learning with function approximation.

Objective	Definition	Comment
value error $\overline{\text{VE}}$	$\lVert \Phi w - V^\pi \rVert_\mu^2$	needs V^π; minimized by Monte Carlo
Bellman error $\overline{\text{BE}}$	$\lVert \delta_w \rVert_\mu^2$	needs double sampling; not learnable from features
projected Bellman error $\overline{\text{PBE}}$	$\lVert \Pi \delta_w \rVert_\mu^2$	zero at the TD fixed point
TD error $\overline{\text{TDE}}$	$\mathbb{E}[\delta_t^2]$	what residual gradient minimizes

The TD fixed point is the zero of $\overline{\text{PBE}}$, so a stochastic gradient method on $\overline{\text{PBE}}$ would converge to it, on- or off-policy. Let the states come from any distribution μ (on- or off-policy), with $D = \text{diag}(\mu)$, and let ρ be the importance-sampling ratio of the action. Since $\Pi^\top D \Pi = D\Phi(\Phi^\top D \Phi)^{-1}\Phi^\top D$,

$$\overline{\text{PBE}}(w) = (\Phi^\top D \delta_w)^\top (\Phi^\top D \Phi)^{-1} (\Phi^\top D \delta_w) = \mathbb{E}[\rho\, \delta\, \phi]^\top\, \mathbb{E}[\phi\phi^\top]^{-1}\, \mathbb{E}[\rho\, \delta\, \phi], \tag{9.4}$$

where $\phi = \phi(s_t)$, $\phi' = \phi(s_{t+1})$, $\delta = r_t + \gamma w^\top \phi' - w^\top \phi$, and the expectations are over the sampled transitions. The gradient of $\mathbb{E}[\rho\delta\phi]$ is $\mathbb{E}[\rho\, \phi(\gamma\phi' - \phi)^\top]$, so

$$-\tfrac{1}{2}\nabla_w \overline{\text{PBE}}(w) = \mathbb{E}\big[\rho\, (\phi - \gamma\phi')\, \phi^\top\big]\, u^*(w), \qquad u^*(w) = \mathbb{E}[\phi\phi^\top]^{-1}\, \mathbb{E}[\rho\, \delta\, \phi].$$

This is a product of expectations, and the product of two independent samples would be needed to estimate it without bias. The way out (Sutton, Maei, et al. 2009) is to learn the second factor separately: $u^*(w)$ is the least-squares solution of predicting $\rho\delta$ from ϕ, which the standard LMS rule $u \leftarrow u + \eta_u\, \rho\, (\delta - u^\top \phi)\, \phi$ tracks. Plugging u in for u^* gives *GTD2*:

$$w \leftarrow w + \eta\, \rho\, (\phi - \gamma\phi')\, (u^\top \phi).$$

Rewriting the gradient with $\mathbb{E}[\phi\phi^\top]u^* = \mathbb{E}[\rho\delta\phi]$ as $\mathbb{E}[\rho\delta\phi] - \gamma\, \mathbb{E}[\rho\, \phi'\phi^\top]\, u^*$ gives *TDC*, "TD with gradient correction": the usual TD update plus a correction term (Exercise 9.13).

Algorithm 9.5 TDC (TD with gradient correction)

Input: target policy π, behavior policy b, features ϕ with $\phi(\text{terminal}) = 0$, step sizes $\eta, \eta_u > 0$, discount γ.

1. Initialize w; set $u \leftarrow 0$.
2. For each transition (s, a, r, s') generated by b:
   1. $\rho \leftarrow \pi(a \mid s)/b(a \mid s)$; $x \leftarrow \phi(s)$; $x' \leftarrow \phi(s')$.
   2. $\delta \leftarrow r + \gamma\, w^\top x' - w^\top x$.
   3. $w \leftarrow w + \eta\, \rho\, \big(\delta\, x - \gamma\, x'\, (u^\top x)\big)$.
   4. $u \leftarrow u + \eta_u\, \rho\, (\delta - u^\top x)\, x$.

Output: $w \approx w_{\mathrm{TD}}$, the zero of $\overline{\mathrm{PBE}}$ under the sampling distribution.

Both methods cost $O(d)$ per step, like TD. Sutton, Maei, et al. (2009) proved that GTD2 and TDC converge with probability one to the TD fixed point, on- or off-policy, for independent samples of transitions, with linear features, bounded second moments, invertible $\mathbb{E}[\phi\phi^\top]$ and A, and step sizes decreasing in the Robbins–Monro manner; for TDC the secondary weights u must adapt on a faster time scale than w ($\eta_t/\eta_{u,t} \to 0$). An earlier algorithm, GTD (Sutton, Szepesvári, et al. 2009), minimized a different norm of the expected update; GTD2 and TDC were faster. Maei (2011) extended the family to eligibility traces, nonlinear function approximation and control (Greedy-GQ).

On Baird's counterexample (Figure 9.5b), TDC behaves as advertised in one sense: the weights stay bounded and settle within a few hundred steps, and the square root of $\overline{\mathrm{PBE}}$ falls from 8.2 to about 0.13 after 1000 steps. (Baird's eight features are linearly dependent on seven states, so the theorem's assumptions do not even hold, yet TDC is stable.) In another sense it is disappointing: after 1000 steps the values are still between 1.76 and 2.15 (about 1.9 on average), and they approach the true value 0 very slowly. Even the noise-free expected TDC update leaves them at about 1.87 after 300,000 iterations. Along the direction of nearly constant value functions, the projected Bellman error is almost flat when γ is close to 1, so its gradient is tiny. Gradient-TD methods are stable, but they can be slow, and the extra step size is one more thing to tune; regularized variants such as TDRC (Ghiassian et al. 2020) reduce the tuning.

9.9.1 Bellman residual minimization and double sampling

Why not drive the Bellman error itself to zero, and do honest gradient descent on $\overline{\mathrm{BE}}$? The obvious stochastic version is Baird's *residual-gradient* algorithm (Baird 1995), which follows the full gradient of the squared TD error of each transition:

$$w \leftarrow w + \eta\,\delta_t\,\big(\phi(s_t) - \gamma\,\phi(s_{t+1})\big).$$

This is true stochastic gradient descent, so it is stable, on- or off-policy. But its objective is $\overline{\mathrm{TDE}}$, not $\overline{\mathrm{BE}}$, and the two differ whenever the next state is random. Conditioning on $s_t = s$,

$$\mathbb{E}[\delta_t^2 \mid s] = \big(\mathbb{E}[\delta_t \mid s]\big)^2 + \mathrm{Var}(\delta_t \mid s) = \delta_w(s)^2 + \mathrm{Var}\big(r_t + \gamma V_w(s_{t+1}) \mid s\big),$$

and the variance term depends on w. Minimizing $\overline{\mathrm{TDE}}$ therefore also tries to make the values of the possible successors of each state similar, so that the targets vary less.

The Trail shows how strong this pull can be. Take the *tabular* Trail under the uniform policy, so that the true values $(1.76, 2.8, 4)$ make the Bellman error zero. The minimizer of $\overline{\mathrm{TDE}}$, weighted by μ, is instead $(1.43, 2.07, 3.24)$ (Exercise 9.14). Even Goal, whose only transition is deterministic, is pulled from 4 down to 3.24: from Middle the two targets are 2 (exit) and 0.9 V(Goal) (go), and lowering V(Goal) makes them more alike.

To estimate the gradient of $\overline{\mathrm{BE}}$ without this bias, one needs two *independent* next states from the same state, $\delta(s, s')\,\big(\phi(s) - \gamma\phi(s'')\big)$, which is possible with a simulator that can be reset to

s, or in a deterministic environment, but not from a single stream of experience. This is the *double-sampling problem*. Even with double sampling, the Bellman error has a deeper defect: it is not a function of the data distribution. Sutton and Barto (2018, sec. 11.6) construct two MDPs that produce exactly the same distribution of feature vectors and rewards but have different $\overline{\text{BE}}$ minimizers, so no algorithm that sees only features and rewards can find the minimizer. The TD fixed point, and $\overline{\text{PBE}}$, are determined by the data. Scherrer (2010) compares the TD fixed point and Bellman residual minimization as two "oblique projections", and finds the TD solution usually better in practice, though less stable in theory.

KEY IDEA

With linear function approximation the right target is the TD fixed point, the zero of the projected Bellman error. On-policy semi-gradient TD reaches it; off-policy, gradient-TD methods such as TDC reach it by true stochastic gradient descent. Minimizing the squared TD error instead gives a biased answer whenever transitions are random.

9.9.2 Emphatic TD ●●●

Emphatic TD (Sutton et al. 2016) takes a different route: keep the simple semi-gradient update, but reweight it so that the weighting of the states makes the expected update matrix positive definite again. A *followon trace* accumulates the discounted, importance-weighted "interest" $i(s) \geq 0$ that the past puts on the current state, $F_t = \rho_{t-1}\gamma F_{t-1} + i(s_t)$, and emphatic TD(0) updates

$$w \leftarrow w + \eta F_t \rho_t \delta_t \phi(s_t).$$

In expectation, the states are then weighted by $f^\top = (\mu \circ i)^\top (I - \gamma P^\pi)^{-1}$, the states that the target policy would reach from the sampled ones, discounted, and the matrix $\Phi^\top \operatorname{diag}(f)(I - \gamma P^\pi)\Phi$ is positive definite for any behavior policy that covers the target policy. Yu (2015) proved convergence with probability one under general conditions. On Baird's counterexample, with interest 1 everywhere, the lower state gets weight $f(7) \approx 99.1$ against $\frac{1}{7}$ for the others, and the expected update converges to the true values. The weakness is variance: F_t is a product of importance ratios and can become enormous (Exercise 9.17).

9.10 Fitted value iteration and fitted Q iteration

So far the agent learned online, one transition at a time. Often we have a *batch* of data instead: a set $\mathcal{D}$ of transitions (s_i, a_i, r_i, s'_i), collected earlier by some behavior, and we want a good policy from it (Chapter 17 treats this setting in full). The cleanest idea is to run value iteration (Algorithm 4.3) and replace each exact Bellman backup by a regression: compute targets with the current estimate, fit a new estimate to them, repeat. With action values this is *fitted Q iteration* (FQI), studied by Ernst et al. (2005) with regression trees and by Riedmiller (2005) with neural networks; the same idea for state values goes back to fitted value iteration (Gordon 1995).

Algorithm 9.6 Fitted Q iteration

Input: a data set $\mathcal{D} = \{(s_i, a_i, r_i, s'_i, \tau_i)\}_{i=1}^{N}$ with $\tau_i = 1$ if s'_i is terminal, a regression method over a function class $\mathcal{F}$, discount γ, number of iterations K.

1. Initialize Q_0 (for example $Q_0 = 0$).
2. For $k = 0, 1, \dots, K-1$:
   1. For each i: $y_i \leftarrow r_i + \gamma\,(1 - \tau_i)\max_{a'} Q_k(s'_i, a')$.
   2. $Q_{k+1} \leftarrow \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \big(f(s_i, a_i) - y_i\big)^2$.

Output: Q_K and its greedy policy $\pi(s) = \arg\max_a Q_K(s, a)$.

Each iteration is an ordinary supervised regression with fixed targets, which can use any regression method and all the data at once. With linear features it is a least-squares problem per action (the library's `fitted_q_iteration`).

Example 9.6 Fitted Q iteration on the aggregated Trail

Give the aggregated Trail one weight per action: $Q(\text{merged}, \text{exit})$, $Q(\text{merged}, \text{go})$ for the shared feature of Start and Middle, and $Q(\text{Goal}, \text{finish})$. Collect data with the uniform policy. Per episode we expect $\frac{1}{2}$ exit and $\frac{1}{2}$ go transitions from Start and $\frac{1}{4}$ of each from Middle, so in every regression Start's transitions count twice as much as Middle's. The exit targets are 1 and 2, which gives $Q(\text{merged}, \text{exit}) = (2 \cdot 1 + 1 \cdot 2)/3 = \frac{4}{3}$ after one iteration. Goal's target is 4. The go targets are $0.9 \max_a Q(\text{merged}, a)$ from Start (whose next state, Middle, has the merged features) and $0.9 \times 4 = 3.6$ from Middle. So FQI iterates

$$Q(\text{merged}, \text{go}) \leftarrow \tfrac{2}{3} \cdot 0.9 \max\big(Q(\text{merged}, \text{go}), \tfrac{4}{3}\big) + \tfrac{1}{3} \cdot 3.6 = 0.6 \max\big(Q(\text{merged}, \text{go}), \tfrac{4}{3}\big) + 1.2,$$

a contraction with fixed point 3.0. The values are wrong, since $Q^*(\text{Start}, \text{go}) = 3.24$ and $Q^*(\text{Middle}, \text{go}) = 3.6$, but $3.0 > \frac{4}{3}$, so the greedy policy goes on in both states and is optimal. The lab finds 1.337, 2.991 and 4.0 from 3000 sampled episodes. With different data the same features can mislead: if the data come from a behavior that rarely goes on at Middle, the go regression is dominated by Start's low bootstrapped targets, and FQI ends up preferring to exit (Exercise 9.16).

♦

How errors propagate. If every regression were exact, FQI would be value iteration and $Q_k \to Q^*$. In reality each iteration makes an error $\epsilon_k = Q_{k+1} - \mathcal{T}^* Q_k$, where $\mathcal{T}^*$ is the Bellman optimality operator. If these errors are bounded in the sup norm, $\|\epsilon_k\|_\infty \le \epsilon$, then the approximate value iteration bound of Theorem 4.11 applies: the greedy policies π_k eventually satisfy

$$\limsup_{k \to \infty} \|Q^* - Q^{\pi_k}\|_\infty \le \frac{2\gamma}{(1-\gamma)^2}\,\epsilon.$$

The errors do not accumulate without limit, because each backup contracts the old errors by γ, but they are amplified: by a factor of 180 for $\gamma = 0.9$. Regression, however, controls a weighted squared error on the data distribution, not the error in every state. Bounds in that language need a measure of how well the data cover the states that the policies being evaluated would reach (Section 9.10.1).

Stability. FQI with an arbitrary function class can diverge, for the same reasons as off-policy TD: a regression step need not be a non-expansion in the sup norm, and the composition of a non-expansive projection with a contraction is only guaranteed to contract in a norm in which both behave. Gordon (1995) identified a class of approximators that are safe: *averagers*, whose prediction at any point is a weighted average of the training targets with nonnegative weights summing to one. State aggregation, nearest neighbors and kernel smoothing with positive kernels are averagers; the fitted Bellman operator is then a γ-contraction in the sup norm, and fitted value iteration converges. This is why aggregation never diverges, and why tree-based FQI uses trees whose leaves average their targets.

From FQI to DQN. FQI is the bridge to deep reinforcement learning. Replace the function class by a neural network. Instead of solving each regression exactly, take a number of stochastic gradient steps on minibatches, toward targets computed with a frozen copy of the network from the previous iteration. And instead of a fixed data set, keep a buffer of recent experience that grows as the agent acts. The result is the deep Q-network of Chapter 10: its *target network* plays the role of Q_k, and its *replay buffer* the role of $\mathcal{D}$.

9.10.1 Concentrability ●●●

The analyses of approximate value iteration in weighted norms (Munos 2007; Munos and Szepesvári 2008) replace the sup norm by an L_p norm under the data distribution ν, and the price is a *concentrability coefficient* C: roughly, a bound on how much the state-action distributions that arise from following sequences of policies from the evaluation distribution ρ can exceed ν, weighted by γ over time. Bounds then take the form

$$\limsup_{k\to\infty} \|Q^* - Q^{\pi_k}\|_{p,\rho} \le \frac{2\gamma}{(1-\gamma)^2}\, C^{1/p} \limsup_{k\to\infty} \|\epsilon_k\|_{p,\nu},$$

and the finite-sample analysis of Munos and Szepesvári (2008) bounds $\|\epsilon_k\|_{p,\nu}$ by an estimation error that shrinks with the amount of data plus the *inherent Bellman error* of the function class, the distance between $\mathcal{F}$ and the image $\mathcal{T}^*\mathcal{F}$. Two lessons follow. First, coverage matters: if the data never visit states that good policies reach, C is infinite and nothing can be guaranteed, which is the central difficulty of offline reinforcement learning (Section 17.7.2). Second, the function class must be *closed* under the Bellman operator, approximately, not merely able to represent Q^*. Chen and Jiang (2019) discuss why such completeness assumptions seem unavoidable for FQI-style methods.

9.11 Theory of reinforcement learning with function approximation ●●●

The question behind this section is: when can an agent learn a near-optimal policy with a number of samples polynomial in the number of features d and the horizon, and independent of the number of states? For tables the answer is well understood (Chapter 6; Chapter 16). With function approximation it depends delicately on the assumptions.

Realizability is not enough. The weakest natural assumption is that Q^* is in the function class, for example $Q^*(s,a) = \phi(s,a)^\top w^*$ for known features. Weisz et al. (2021) showed that this does not suffice even for planning with a generative model, which can be queried at any state: any planner that reliably finds a near-optimal action must make a number of queries exponential in d, or in the horizon H if that is smaller, in the fixed-horizon setting (and exponential in d in the discounted setting). Du et al. (2020) showed that approximate realizability with a small misspecification is not sufficient either, and Wang et al. (2021a) proved exponential lower bounds for offline evaluation with linear realizability even under good coverage. The intuition shared by these constructions is that knowing the form of Q^* does not tell you how to propagate information through the Bellman equation: the backups of functions in the class leave the class, and the agent cannot tell which of exponentially many hypotheses is consistent with its data.

What suffices. Stronger, structural assumptions restore polynomial guarantees. *Bellman completeness*, that the class is closed under the Bellman operator, underlies the analyses of FQI and of online algorithms for classes with low inherent Bellman error (Zanette et al. 2020). *Linear MDPs*, in which transition probabilities and rewards are themselves linear in known features, admit exploration algorithms with regret polynomial in d and the horizon (Chapter 16). General complexity measures, such as Bellman rank and the Bellman eluder dimension, unify many such cases (Section 24.6). All of these assume more than realizability, and none of them is easy to check for a given problem.

Where do features come from? In practice the features are learned. A deep network trained only on its TD loss learns features that serve its current targets, and those can be poor: bootstrapping can collapse the rank of the learned features (Kumar et al. 2021). Adding auxiliary prediction tasks (Jaderberg, Mnih, et al. 2017) or self-predictive objectives (Schwarzer et al. 2021) often helps. Older work built features from the structure of the state space, such as the eigenvectors of the graph Laplacian (Mahadevan and Maggioni 2007), and Bellemare et al. (2019) characterize optimal representations as those that approximate well a particular set of extremal value functions. A theory that explains which learned representations make reinforcement learning efficient does not yet exist.

RESEARCH NOTE

Open questions about representations: Which properties of learned features (rank, smoothness, closure under the Bellman operator, predictiveness of future features) are necessary or sufficient for efficient TD learning? Can a representation be learned once, from reward-free data, and reused for many tasks without the exponential lower bounds above applying? How should representation learning interact with exploration? Exercise 9.19 is a starting point.

9.12 Summary

- When states are too many to tabulate, values are represented by a function V_w with few parameters, which generalizes from visited states to similar ones. Errors are traded off by the on-policy distribution μ in the objective $\overline{\text{VE}}$.
- Linear functions of features are the best-understood case. State aggregation, polynomials, Fourier bases, radial basis functions and tile coding are standard constructions; tile coding gives sparse, local features that make linear methods practical in a few continuous dimensions.
- Gradient Monte Carlo is stochastic gradient descent on $\overline{\text{VE}}$ and converges to the best approximation. Semi-gradient TD bootstraps, ignores the dependence of its target on w, and converges on-policy to a different point, the TD fixed point.
- The TD fixed point solves the projected Bellman equation $\Phi w = \Pi \mathcal{T}^\pi \Phi w$, equivalently $Aw = b$. On the aggregated Trail it is 1.81 for the merged state, against 2.11 for Monte Carlo. (●●) On-policy, $\Pi\mathcal{T}^\pi$ is a γ-contraction in $\|\cdot\|_\mu$, linear TD converges, and its error is at most $1/\sqrt{1-\gamma^2}$ times the best possible.
- LSTD solves for the TD fixed point directly, without a step size, at $O(d^2)$ cost per step; LSPI reuses one batch of data for policy iteration.
- Semi-gradient SARSA with tile coding solves control problems such as Mountain Car, though its convergence theory is incomplete.
- The deadly triad, function approximation with bootstrapping and off-policy data, can make learning diverge, as Baird's counterexample shows. (●●) Gradient-TD methods such as TDC restore convergence by descending the projected Bellman error; minimizing the squared TD error instead is biased by the double-sampling problem.
- Fitted Q iteration turns value iteration into repeated regression on a batch; its errors are amplified by about $2\gamma/(1-\gamma)^2$, and its guarantees need coverage of the data. With a neural network, a target network and a replay buffer, it becomes DQN.

The next chapter takes that step: deep Q-networks, the engineering that made them work on Atari games, and the many improvements that followed.

9.13 Further Reading

Sutton and Barto (2018, ch. 9–11) is the standard treatment of everything in the ● sections, including tile coding, the deadly triad and gradient-TD methods, with many further examples; Chapter 11's discussion of which objectives are learnable is particularly worth reading. Bertsekas and Tsitsiklis (1996) remains the classic source on approximate dynamic programming and its error bounds, and Szepesvári (2010) gives a compact, rigorous account of linear TD, LSTD and gradient TD. For a comparison of the many policy-evaluation algorithms, with experiments, see the survey of Dann et al. (2014); Geist and Pietquin (2013) organize them by the objective they optimize.

The convergence theory of linear TD is due to Tsitsiklis and Van Roy (1997), a paper that is still the best place to learn how such proofs work, including the counterexample with nonlinear approximation. LSTD comes from Bradtke and Barto (1996) and Boyan (2002), LSPI from

Lagoudakis and Parr (2003). Baird's counterexample and the residual-gradient algorithm are in Baird (1995). Gradient-TD methods began with Sutton, Szepesvári, et al. (2009) and Sutton, Maei, et al. (2009); Maei (2011)'s thesis develops the full family. Emphatic TD is introduced in Sutton et al. (2016). The Fourier basis is from Konidaris et al. (2011), and Sutton (1996) is a short and influential demonstration of tile coding on control problems.

Fitted value iteration with averagers is in Gordon (1995); fitted Q iteration in Ernst et al. (2005) and Riedmiller (2005). The error analysis in L_p norms, with concentrability coefficients, is developed by Munos (2007) and Munos and Szepesvári (2008).

(●●●) For the theory of reinforcement learning with function approximation, the lecture notes of Agarwal et al. (2022) are the most accessible entry point. The lower bounds of Weisz et al. (2021), Du et al. (2020) and Wang et al. (2021a) show what realizability alone cannot buy; Chen and Jiang (2019) discuss the role of completeness in batch learning. For representation learning, see Bellemare et al. (2019) for a geometric view, Kumar et al. (2021) for how bootstrapping degrades learned features, and Hasselt et al. (2018) for an empirical study of the deadly triad in deep Q-learning.

9.14 Exercises

Exercise 9.1 ● The aggregated Trail with another policy

Consider the aggregated Trail of Example 9.1 with $\gamma = 0.9$ and the policy $p_0 = 0.8$, $p_1 = 0.5$. (a) Compute the true values, the expected numbers of visits η and the on-policy distribution μ. (b) Compute the Monte Carlo solution. (c) Compute the TD fixed point, by writing the condition that the expected update vanishes for each feature. (d) Which of the two is closer to $V^\pi(\text{Start})$, and which to $V^\pi(\text{Middle})$?

Exercise 9.2 ● Projection with another weighting

For the aggregated Trail, compute the projection matrix Π when the states are weighted uniformly, $\mu = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, instead of by the on-policy distribution. (a) Compute ΠV^π for the uniform policy and check that $\Pi^2 = \Pi$. (b) Compute the TD fixed point for this weighting. (c) Which states does each weighting favor, and why might the on-policy weighting be preferable?

Exercise 9.3 ● Semi-gradient TD by hand

On the aggregated Trail, start from $w = (0, 0)$ and apply semi-gradient TD(0) with $\eta = 0.5$ and $\gamma = 0.9$ to two episodes: first Start, go, Middle, go, Goal, finish; then Start, go, Middle, exit. Give the TD error and the weights after every update. Which updates change w_1, and why do the first two updates of the first episode change nothing?

Exercise 9.4 ● Tile coding: resolution and generalization

A one-dimensional state $x \in [0,1]$ is tile-coded with n tilings, each made of tiles of width ω, offset from each other by ω/n. (a) How many features are active for each state, and how finely can the approximate value function change? (b) For two states at distance $\Delta < \omega$, how many active tiles do they share, approximately? What does this say about how an update at one state affects the other? (c) Show that with step size $\eta = \alpha/n$ a single update moves the value of the updated state by α times its TD error. (d) What changes if you double n but keep ω, and what if you halve ω but keep n?

Exercise 9.5 • Choosing features

For each problem, propose features for a linear value function and explain your choice: (a) CartPole, with state $(x, \dot{x}, \theta, \dot{\theta})$; (b) a stock of inventory between 0 and 1000 units, where the value depends smoothly on the stock; (c) a grid world with 100×100 cells and walls; (d) a board game such as tic-tac-toe. In which cases would you prefer tile coding, a Fourier basis or aggregation, and in which would you prefer a neural network?

Exercise 9.6 • The $w \to 2w$ example

In the two-state example of Section 9.8, the second state's transition ends the episode with reward 0. (a) Show that an update of the first state multiplies w by $1 + \eta(2\gamma - 1)$ and an update of the second state multiplies it by $1 - 4\eta$. (b) If the data contain only the first transition, for which γ does w diverge? (c) On-policy, every episode updates both states, in order. For which η does w converge to 0 for every $\gamma \in [0,1)$? (d) Which leg of the deadly triad is removed by each of these changes: using the Monte Carlo return as the target; giving each state its own weight; following the on-policy distribution?

Exercise 9.7 • Find the legs of the triad

For each method, say which of the three legs of the deadly triad it has, and whether the convergence results of this chapter apply: (a) tabular Q-learning; (b) semi-gradient SARSA with tile coding, following its own ε-greedy policy; (c) gradient Monte Carlo with a neural network; (d) linear Q-learning trained on transitions stored from many past policies; (e) LSTD evaluating the behavior policy that generated its data; (f) fitted Q iteration with state aggregation.

Exercise 9.8 • Fourier features on the random walk (programming)

Extend Task 3 of Lab 9.1. (a) Plot the RMS error of the Monte Carlo solution and of the TD fixed point against the order of the Fourier basis, from 1 to 20, on a log scale. Why do orders 1 and 2 give the same error? (b) Learn the order-5 weights with semi-gradient TD(0), using the per-feature step sizes $\eta/\|c\|$, for 5000 episodes and two values of η. How close do you get to the TD fixed point, and how does that compare with LSTD from 300 episodes?

Exercise 9.9 •• Deriving the TD fixed point

(a) Derive the expected update $\mathbb{E}[\delta_t \phi(s_t)] = b - Aw$ of linear TD(0) under the on-policy distribution, with A and b as in (9.2), stating where you use the Markov property. (b) Show that $Aw = b$ holds if and only if $\Phi w = \Pi \mathcal{T}^\pi \Phi w$. (c) Write down the corresponding A and b for linear TD with Q-values, $\delta_t = r_t + \gamma\, w^\top \phi(s_{t+1}, a_{t+1}) - w^\top \phi(s_t, a_t)$, on-policy.

Exercise 9.10 ●● The contraction in episodic tasks

Let P^π be the sub-stochastic transition matrix among the non-terminal states of an episodic task in which every episode ends with probability one, and let μ be the on-policy distribution of Definition 9.1. (a) Show that $\sum_s \mu(s) P^\pi(s' \mid s) \le \mu(s')$ for every s'. (b) Deduce that $\|P^\pi v\|_\mu \le \|v\|_\mu$, adapting step 1 of the proof of Lemma 9.3 (careful: the rows of P^π may sum to less than one). (c) Conclude that the lemma and Theorem 9.5 hold for episodic tasks with $\gamma < 1$, and verify the bound on the aggregated Trail.

Exercise 9.11 ●● Two error bounds

(a) Using only the triangle inequality and Lemma 9.3, show that $\|\Phi w_{\mathrm{TD}} - V^\pi\|_\mu \le \frac{1}{1-\gamma} \|\Pi V^\pi - V^\pi\|_\mu$. (b) Explain why the bound of Theorem 9.5 is sharper, and by how much for $\gamma = 0.9$ and $\gamma = 0.99$. (c) Evaluate both bounds on the aggregated Trail and compare them with the actual error.

Exercise 9.12 ●● The Sherman–Morrison update in LSTD

(a) Prove the Sherman–Morrison formula: if M is invertible and $1 + v^\top M^{-1} u \neq 0$, then $(M + uv^\top)^{-1} = M^{-1} - \frac{M^{-1} u\, v^\top M^{-1}}{1 + v^\top M^{-1} u}$. (b) Show that step 2.3 of Algorithm 9.3 applies it with $u = x$ and $v = x - \gamma x'$. (c) Count the operations per step. (d) Why must the algorithm start from εI rather than from 0, and what is the effect of ε on the solution after t steps?

Exercise 9.13 ●● The gradient of the projected Bellman error

(a) Starting from $\overline{\mathrm{PBE}}(w) = \|\Pi \delta_w\|_\mu^2$, derive the expectation form (9.4). (b) Derive $-\frac{1}{2} \nabla_w \overline{\mathrm{PBE}}$ and the GTD2 update. (c) Derive the TDC update and show that, when $u = u^*(w)$, its expected value equals $-\frac{1}{2} \nabla_w \overline{\mathrm{PBE}}$. (d) Why is the gradient a product of two expectations, and why can it not be estimated without bias from a single transition?

Exercise 9.14 ●● The double-sampling bias on the Trail

For the tabular Trail under the uniform policy with $\gamma = 0.9$ and the weights $\mu = (\frac{4}{7}, \frac{2}{7}, \frac{1}{7})$: (a) write the mean squared TD error $\overline{\mathrm{TDE}}$ as a function of $V(\mathrm{Start})$, $V(\mathrm{Middle})$ and $V(\mathrm{Goal})$; (b) find its minimizer by solving the linear equations that make its gradient vanish; (c) show that it differs from V^π and explain the direction of each difference; (d) show that the expected residual-gradient update $\mathbb{E}[\delta_t(\phi_t - \gamma \phi_{t+1})]$ is $-\frac{1}{2} \nabla \overline{\mathrm{TDE}}$, not $-\frac{1}{2} \nabla \overline{\mathrm{BE}}$, and give an unbiased estimator of $\nabla \overline{\mathrm{BE}}$ that uses two independent next states.

Exercise 9.15 ●● The futility of discounting

Let μ_π be the stationary distribution of π in a continuing finite MDP and $\gamma < 1$. (a) Prove that $\sum_s \mu_\pi(s) V_\gamma^\pi(s) = r(\pi)/(1-\gamma)$, with $r(\pi)$ the average reward. (b) Conclude that ranking policies by their average discounted value under their own stationary distribution is the same as ranking them by average reward, whatever γ. (c) Why does this argument need function approximation to have any force, that is, why is it harmless in the tabular case?

Exercise 9.16 ●● When fitted Q iteration picks the wrong policy

Repeat Example 9.6 with data from a behavior policy with $b(\text{go} \mid \text{Start}) = p_0$ and $b(\text{go} \mid \text{Middle}) = p_1$, in the expected proportions. (a) Show that the exit regression gives $Q(\text{merged}, \text{exit}) = \frac{(1-p_0)+2p_0(1-p_1)}{(1-p_0)+p_0(1-p_1)}$. (b) Show that, if go is preferred at the fixed point, $Q(\text{merged}, \text{go}) = \frac{3.6\,p_1}{0.1+p_1}$. (c) For $p_0 = 0.5$, find the value of p_1 below which the greedy policy of FQI exits, although going on is optimal. (d) What does this say about the data FQI needs?

Exercise 9.17 ●●● Emphatic TD on Baird's counterexample

(a) Implement emphatic TD(0) with interest $i(s) = 1$ on Baird's counterexample, both in expectation (using the weighting f of Section 9.9.2) and from sampled transitions. (b) Compute f and the eigenvalues of the emphatic matrix $\Phi^\top \operatorname{diag}(f)(I - \gamma P^\pi)\Phi$, and compare them with the eigenvalues of the off-policy TD matrix. (c) Run the sampled version for several seeds and report the largest followon trace and the final weights. What does this say about the practicality of emphatic methods, and what has been proposed to reduce their variance?

Exercise 9.18 ●●● Why realizability is not enough

Read Weisz et al. (2021) (and, for contrast, the definition of a linear MDP in Chapter 16). (a) State precisely what is assumed and what is proved. (b) Explain in words why knowing that Q^* is linear in known features does not let a planner propagate information through the Bellman equation. (c) Which additional assumption makes fitted Q iteration with linear features work, and why does a linear MDP satisfy it?

Exercise 9.19 ●●● Representation learning for reinforcement learning

Write a two-page research proposal on learning features for value-based reinforcement learning. Survey three approaches (for example auxiliary tasks, self-predictive representations and Laplacian-based features), state what each is meant to guarantee, and design a small experiment, on the random walk, Mountain Car or a grid world, that would distinguish features that are good for TD learning from features that are merely good for reconstruction or prediction.

CHAPTER 10

Deep Value-Based Reinforcement Learning

WHAT YOU WILL LEARN

- Explain how a deep Q-network (DQN) turns fitted Q iteration into an online algorithm, and what experience replay and target networks each contribute.
- Implement DQN with a small neural network, train it on CartPole, and read its learning curves and value estimates critically.
- Explain why Q-learning overestimates action values and compute Double DQN targets.
- Describe dueling networks, prioritized replay, multi-step targets, noisy networks and Rainbow, and what the ablation studies say about them.
- State the distributional Bellman equation and its contraction property, and carry out the C51 projection and the quantile-regression update by hand (●●).
- Survey how value-based agents were scaled up and made data-efficient, and the open problems of plasticity and replay ratio (●●●).
- Apply a checklist of hyperparameters, debugging steps and evaluation practices to a deep value-based agent.

In 2013 a single program learned to play several Atari video games from the raw pixels on the screen and the score, with no knowledge of the games' rules; two years later the same method, with the same network and the same settings, played 49 games, on many of them at the level of a professional human tester (Mnih et al. 2013, 2015). The method was Q-learning (Algorithm 6.3) with a deep neural network as its function approximator, plus two stabilizing devices, and it started the field of deep reinforcement learning.

Chapter 9 explains why this was not an obvious success. A neural network trained by Q-learning has all three legs of the deadly triad (Section 9.8): function approximation, bootstrapping, and off-policy data. Early attempts to combine Q-learning with neural networks had been fragile, and theory offered no guarantee. The deep Q-network (DQN) made the combination work in practice, by making Q-learning look as much as possible like the fitted Q iteration of Section 9.10: a sequence of supervised regression problems with fixed targets and plentiful, well-mixed data.

This chapter builds DQN piece by piece (Section 10.1 and Section 10.2), runs it on CartPole with the small NumPy network of the labs (Section 10.3), and measures what each piece does. It then covers the improvements that followed: Double DQN against overestimation (Section 10.4), dueling networks (Section 10.5), prioritized replay (Section 10.6), and multi-step targets, noisy networks and their combination in Rainbow (Section 10.7). Section 10.8 (●●) learns the whole

distribution of the return instead of its mean. Section 10.9 (●●●) surveys distributed and data-efficient agents and the open problems they raised. Section 10.10 collects practical advice. A first reading can skip Sections 10.8 and 10.9.

10.1 From fitted Q iteration to deep Q-networks

A neural network is a natural candidate for Q when the state is high-dimensional: the input is the state, for instance a stack of screen images, and the output is a vector with one value per action, $Q_w(s,\cdot) \in \mathbb{R}^{|\mathcal{A}|}$, where w denotes all the weights. One forward pass then gives the values of all actions, so both the greedy action and the target $\max_{a'} Q_w(s',a')$ cost a single evaluation. The network learns its own features (Section 9.2.5); no tile coder or Fourier basis has to be designed.

The idea is old. Lin (1992) trained neural networks with Q-learning, and introduced replaying past experience; TD-Gammon learned backgammon with a neural network and TD learning (Tesauro 1995); neural fitted Q iteration (Riedmiller 2005) trained a network on a batch of transitions by repeated regression. But naive online Q-learning with a neural network, one gradient step on each new transition, tends to be unstable. Three things go wrong at once. Consecutive transitions are highly correlated, so the network fits whatever part of the state space it is currently in and forgets the rest. The target $r+\gamma \max_{a'} Q_w(s',a')$ moves every time w moves, so the regression chases its own tail. And the data come from a policy that keeps changing, which is the off-policy leg of the triad.

Fitted Q iteration avoids all three. Its regressions use a whole data set at once, so no region is forgotten; its targets are computed with the previous iterate and held fixed; and it treats the data as a fixed batch. DQN keeps these properties approximately while learning online. It stores experience in a *replay buffer* and trains on random minibatches drawn from it, which makes the data look like a well-shuffled batch; and it computes targets with a *target network*, a copy of the Q-network that is updated only every C steps, which makes each stretch of C steps a regression with fixed targets, one iteration of fitted Q iteration solved approximately by stochastic gradient descent.

HISTORICAL NOTE

DQN was introduced by Mnih et al. (2013), a workshop paper that learned seven Atari games, and published in full by Mnih et al. (2015), with the target network and an evaluation on 49 games of the Arcade Learning Environment (Bellemare et al. 2013). Experience replay is due to Lin (1992), twenty years earlier. The paper's lesson, that a few engineering choices can make an unstable combination work at scale, shaped the next decade of the field.

10.2 The DQN algorithm

DQN learns $Q_w \approx Q^*$ by minimizing, over minibatches B of transitions (s,a,r,s',τ) sampled from the replay buffer $\mathcal{D}$, the loss

$$L(w) = \frac{1}{|B|} \sum_{(s,a,r,s',\tau)\in B} \ell\big(y - Q_w(s,a)\big), \qquad y = r + \gamma\,(1-\tau) \max_{a'} Q_{\bar{w}}(s',a'), \tag{10.1}$$

where $\tau = 1$ if s' is terminal and 0 otherwise, ℓ is the squared error or the Huber loss (Section 10.2.3), and $\bar{w}$ are the weights of the target network. The target y is treated as a constant: no gradient flows through it, which makes this a semi-gradient method in the sense of Section 9.3. In words: the online network Q_w is regressed toward one-step Q-learning targets computed by a frozen copy of itself.

Algorithm 10.1 Deep Q-network (DQN)

Input: Q-network Q_w, replay capacity N, minibatch size $|B|$, target period C, step size η, exploration schedule ε_t, discount γ, number of steps before learning starts.

1. Initialize w randomly; set $\bar{w} \leftarrow w$; let $\mathcal{D}$ be an empty buffer of capacity N.
2. Observe the initial state s.
3. For $t = 0, 1, 2, \ldots$:
   1. With probability ε_t choose a random action a; otherwise $a \leftarrow \arg\max_{a'} Q_w(s, a')$.
   2. Take a; observe r, s', and whether the episode terminated ($\tau = 1$) or was truncated.
   3. Store (s, a, r, s', τ) in $\mathcal{D}$, overwriting the oldest transition when $\mathcal{D}$ is full.
   4. If learning has started:
      1. Sample a minibatch B uniformly from $\mathcal{D}$.
      2. For each transition in B: $y \leftarrow r + \gamma\,(1 - \tau) \max_{a'} Q_{\bar{w}}(s', a')$.
      3. Take one gradient step on $L(w)$ of (10.1) (for instance with Adam, step size η).
      4. Every C gradient steps: $\bar{w} \leftarrow w$.
   5. If the episode ended (terminated or truncated): reset and observe a new s; else $s \leftarrow s'$.

Output: $Q_w \approx Q^*$ and its greedy policy.

A few steps deserve comment. The greedy action in step 3.1 uses the online network, so the agent acts on its latest knowledge. Step 3.3 stores the termination flag but not the truncation flag: when a time limit cuts an episode, s' still has a value, and the target must bootstrap from it (Section 6.8). In step 3.4.3 only the entries $Q_w(s, a)$ of the actions actually taken receive a gradient; the other outputs of the network are left alone. Figure 10.1 shows how the pieces fit together.

10.2.1 Experience replay

The *replay buffer* holds the last N transitions, a million in the Atari experiments, and each gradient step uses a minibatch sampled uniformly from it (Lin 1992). Replay does three things.

It *decorrelates* the data. Consecutive states of a game or of a cart and pole are nearly identical, and a minibatch of consecutive transitions is a minibatch of near-duplicates. A random minibatch from a large buffer mixes many episodes and many regions of the state space; with 50,000 transitions from episodes of 200 steps, a minibatch of 64 contains on average about eight pairs of transitions from the same episode out of 2016 pairs (Exercise 10.1), instead of all of them.

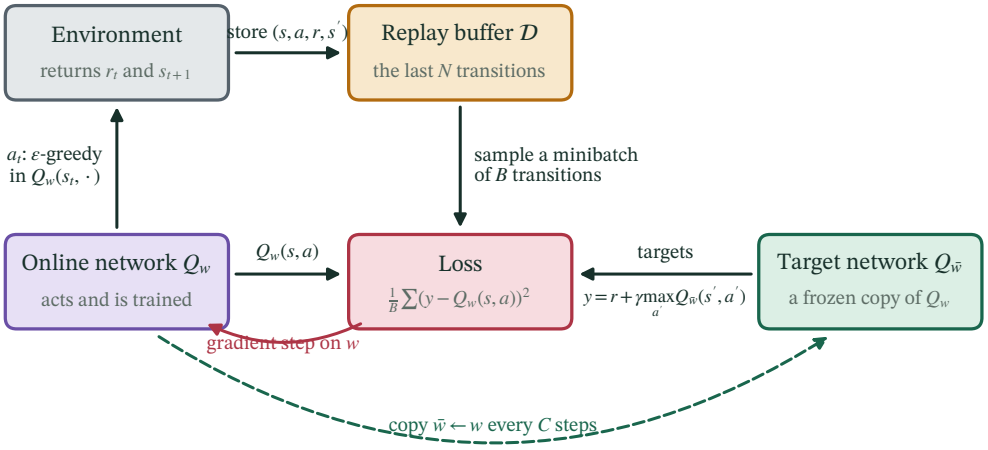


Figure 10.1. The DQN agent. The online network Q_w chooses actions (ε-greedily) and is trained; transitions go into the replay buffer, from which minibatches are sampled; the frozen target network $Q_{\bar{w}}$ computes the targets y; the gradient of the loss updates w; and every C steps w is copied into $\bar{w}$.

It *reuses* data. Each transition is replayed many times, on average about as many times as the minibatch size divided by the number of environment steps per gradient step, so expensive experience is used efficiently. The number of gradient updates per environment step, the *replay ratio*, is an important hyperparameter (DQN's Atari setting used one minibatch of 32 every 4 steps).

It *smooths* the data distribution. The buffer contains transitions from many past policies, so a sudden change in the policy does not immediately change the training distribution. This is also the price of replay: the data are off-policy, generated by older and more exploratory policies than the current greedy one. Q-learning's target does not need the data to be on-policy, which is why DQN uses Q-learning and not SARSA, but the off-policy leg of the deadly triad is now firmly present.

10.2.2 Target networks

If the targets are computed with the online weights w, every gradient step changes the targets of all states that the network generalizes to, including the state just updated. The regression is then chasing a target that moves as it is approached, the feedback loop that makes the deadly triad dangerous. DQN breaks the loop with a *target network*: a copy $\bar{w}$ of the weights that is held fixed for C gradient steps and then overwritten with w. During those C steps the targets are fixed, and the online network solves, approximately, the regression

$$\min_{w} \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\big(r + \gamma \max_{a'} Q_{\bar{w}}(s', a') - Q_w(s, a)\big)^2\Big],$$

one iteration of fitted Q iteration (Algorithm 9.6) with $Q_k = Q_{\bar{w}}$. DQN is fitted Q iteration in which each regression is only partially solved, by C stochastic gradient steps, on a data set that keeps changing.

This view explains the trade-off in choosing C. A large C makes each regression better solved and the targets more stable, but values can then propagate only about one step per target update:

if the rewards are at most 1 and the network starts near zero, after K target updates the values can hardly exceed $1+\gamma+\cdots+\gamma^K$. In our CartPole runs with $C = 500$, about 38 target updates in 20,000 steps left the values near 35, far below the true discounted value of about 93 (Exercise 10.2). A small C propagates values fast but brings back some of the instability. DQN used $C = 10{,}000$ on Atari; our CartPole agent uses $C = 100$.

An alternative to periodic copies is *Polyak averaging*, $\bar{w} \leftarrow (1-\tau)\,\bar{w} + \tau\, w$ after every step with a small τ, which makes the target network an exponential moving average of the online network. It is the standard choice in the actor-critic methods of Chapter 13 (Algorithm 13.1).

10.2.3 The loss, the rewards and the inputs

The Huber loss. DQN clipped the TD error $\delta = y - Q_w(s,a)$ to $[-1,1]$ in its gradient. That is the gradient of the *Huber loss* (Huber 1964)

$$\ell_\kappa(\delta) = \begin{cases} \frac{1}{2}\delta^2 & |\delta| \le \kappa, \\ \kappa\left(|\delta| - \frac{1}{2}\kappa\right) & |\delta| > \kappa, \end{cases} \qquad \ell'_\kappa(\delta) = \mathrm{clip}(\delta, -\kappa, \kappa),$$

with $\kappa = 1$: quadratic for small errors, linear for large ones, so that a few large errors (from a bad target or a rare event) cannot produce huge gradient steps. The Huber loss and reward clipping belong together. In problems where rewards are not clipped and values are large, large errors are the normal state of affairs early in training, and the clipped gradient can slow learning: on CartPole, with a reward of +1 per step and values near 100, our agents learned markedly more slowly with the Huber loss than with the squared error: over five seeds they needed between 7000 and 15,500 steps to pass an average return of 150, against 6000 to 7000, and finished near 200 instead of 270 (Exercise 10.8).

Reward clipping. On Atari, DQN clipped every reward to $[-1,1]$, so that one learning rate could serve games whose scores differ by orders of magnitude. This changes the problem: the agent no longer distinguishes a reward of 10 from one of 1000, and can prefer many small rewards to one large one. It is a pragmatic choice, not a neutral one.

Inputs. A single Atari frame is not a Markov state, since it shows positions but not velocities. DQN converted frames to 84×84 gray-scale images, took the maximum over two consecutive frames to remove flicker, repeated each action for four frames (*frame skipping*), and fed the network the last four processed frames, a *frame stack* that makes the input approximately Markov. The network had three convolutional layers and a fully connected layer of 512 units, with one output per action (up to 18).

Exploration. The exploration rate decayed linearly from $\varepsilon = 1$ to 0.1 over the first million frames and stayed there; evaluation used $\varepsilon = 0.05$. Learning started after 50,000 frames of random play, so that the first minibatches came from a reasonably large buffer. Other settings of the Atari agent: buffer of one million transitions, minibatches of 32, one update every 4 actions, $C = 10{,}000$, $\gamma = 0.99$, and the RMSProp optimizer with a step size of 2.5×10^{-4} (Mnih et al. 2015).

PITFALL

Many DQN bugs live in the target. Check that terminal transitions do not bootstrap ($\tau = 1$), that time-limit truncations do, that the target is computed with the target network and carries no gradient, and that the action index used to pick $Q_w(s, a)$ out of the network's output is the action that was actually taken. A target network that is accidentally the online network, or a mask with the wrong sign, can let a DQN learn something for a while before it falls apart.

10.2.4 Why each piece is needed

DQN combines all three legs of the deadly triad, so it has no convergence guarantee. Its two devices attack the instability from two sides. The target network slows the feedback from the weights to the targets. Replay makes the training distribution broad and slowly changing, which reduces both forgetting and the correlation between successive updates; but by keeping old data it makes training more off-policy. Mnih et al. (2015) reported ablations on five games in which removing either replay or the target network reduced performance substantially, with replay usually the more important of the two. Hasselt et al. (2018) studied many variants systematically and found that value estimates frequently grow far beyond any possible true value for a while and then recover, a phenomenon they called *soft divergence*, while unbounded divergence was rare; target networks, Double DQN (Section 10.4) and multi-step targets (Section 10.7) reduced it, and stronger prioritization (Section 10.6) increased it. The next section reproduces some of this on a small scale.

10.3 DQN on CartPole, step by step

CartPole is the right size for a laptop: a pole balanced on a cart, a state of four numbers (cart position and velocity, pole angle and angular velocity), two actions (push left or right), a reward of $+1$ per step, and an episode that terminates when the pole falls past 12 degrees or the cart leaves the track, and is truncated after 500 steps. With $\gamma = 0.99$ no value can exceed $1/(1-\gamma) = 100$, and a policy that balances to the time limit has a discounted return of 99.3 from the start.

Our agent (`rlcourse.dqn.train_dqn`, with the defaults of `DQNConfig`) uses a network with one hidden layer of 128 ReLU units, built with the NumPy library of Appendix C (Section C.7); Adam (Kingma and Ba 2015) with step size $\eta = 10^{-3}$; minibatches of 64 from a buffer of 50,000 transitions; one gradient step per environment step, starting after 1000 steps; a target network copied every $C = 100$ gradient steps; ε decaying linearly from 1 to 0.05 over 10,000 steps; and the squared loss, with gradients clipped to norm 10. These settings came from a small search over a dozen alternatives; Section 10.10 discusses what mattered. Training for 20,000 steps takes a few seconds.

Figure 10.2 shows what learning looks like. For the first few thousand steps the agent acts almost at random and the pole falls after about twenty steps. As ε decreases and the values sharpen, the returns climb quickly: in every run the average passes 150 between 6000 and 7000 steps. After that, the curves do not settle. Single runs drop by a hundred or more and recover. This noise has several sources: the training returns include exploratory actions; a change in the greedy action in one region of the state space changes which states are visited, and hence what the network is trained on; and a few unlucky updates can make the network forget how to handle states it

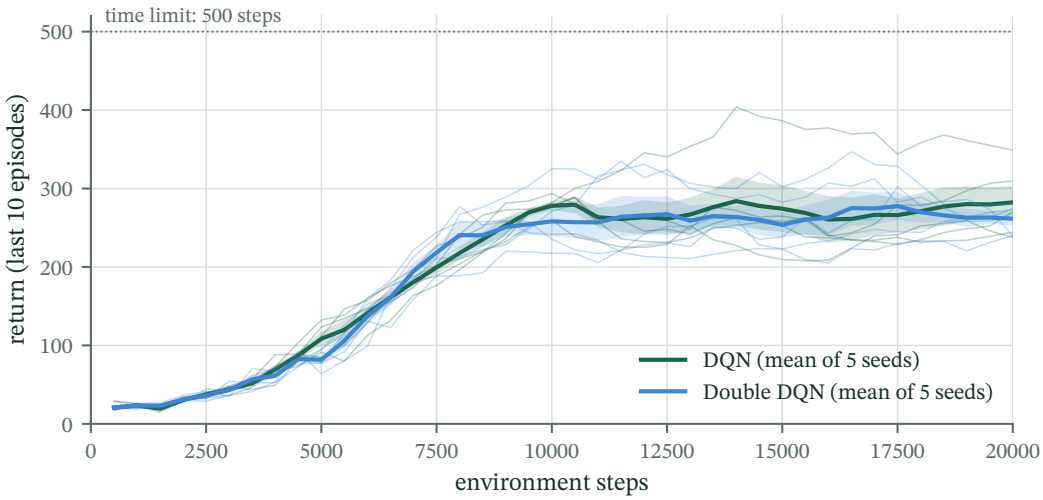


Figure 10.2. DQN and Double DQN on CartPole: the average return of the last ten training episodes, for five seeds each (thin lines) and their mean with a band of ±1 standard error. Returns stay low while ε is large, rise steeply between 5000 and 9000 steps, and then fluctuate between about 200 and 350; individual runs dip and recover. On this problem the two methods are indistinguishable.

rarely sees, such as the cart near the edge of the track. Learning curves from a single seed say little; we always plot several.

What each piece contributes. Figure 10.3 switches the stabilizers off one at a time. To switch replay off we train each step on the 64 most recent transitions, so that each transition is still used about 64 times but the data are recent and consecutive. Without the target network the targets are computed with the online network itself.

Without the target network, learning is fast at first, because values propagate immediately, and the returns peak around 12,000 steps, but then the value estimates run away: in eight runs of 25,000 steps, the final estimates at the start states ranged from −157 to 2205, and the average return fell from about 340 at the peak to about 170. This is soft divergence, the deadly triad at work. Without replay, learning is slower, about 150 instead of 200 at 7500 steps averaged over the same eight seeds, but the value estimates stay close to 100, and by 25,000 steps the returns are as good as with replay. On this small problem the target network matters more than replay; on Atari, where states are images and episodes are long, Mnih et al. (2015) found replay to be the more important of the two. In our runs, dropping both devices kept the estimates near 100 but made the returns very erratic, which suggests that it is the combination of a moving target with old, off-policy data that does the damage (Exercise 10.15).

LAB

Lab 10.1: Deep Q-networks on CartPole (`labs/ch10_dqn.py`, about 60 minutes; the script runs in about a minute). Run `python3 labs/ch10_dqn.py`. You will compute DQN and Double DQN targets by hand, train a DQN, compare its value estimates with the returns it actually obtains, switch off replay and the target network, check that a sum tree samples in proportion to priorities, build n-step transitions, train a “Rainbow-lite” agent, and learn the return distribution of the Trail.

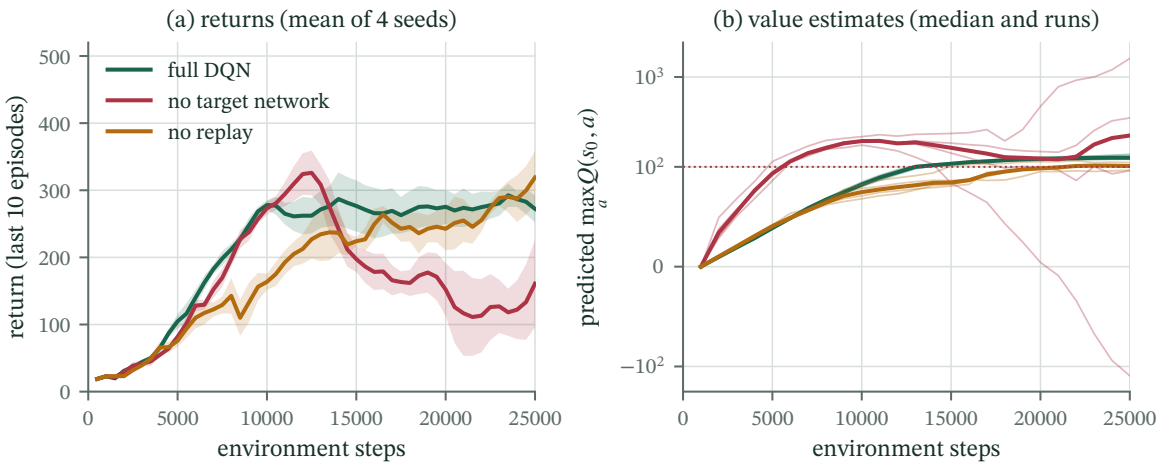


Figure 10.3. What replay and the target network do on CartPole (four seeds each). (a) Average return of the last ten episodes, mean and ±1 standard error. (b) The predicted value $\max_a Q_w(s_0, a)$ of ten fixed start states, on a scale that is linear up to 100 and logarithmic beyond; thin lines are individual runs, thick lines the median. No true value exceeds 100 (dotted). Without the target network, the estimates drift far beyond any possible value, or below zero, and the returns fall after an early peak. Without replay, learning is slower but the estimates stay sensible.

10.4 Overestimation and Double DQN

Q-learning's target uses $\max_{a'} Q(s', a')$, and the maximum of noisy estimates is biased upward. Section 6.7 showed the effect in tables, where the noise comes from sampling. With function approximation there is a second source of noise, the approximation error itself, which is present even with infinite data; Thrun and Schwartz (1993) pointed out early that it leads to systematic overestimation in Q-learning with function approximation.

The size of the effect is easy to compute. Suppose all m actions are equally good, $Q(s', a) = 0$, and the estimates carry independent errors with standard deviation σ. The expected maximum of the estimates is σ times the expected maximum of m standard normal variables: $0.56\,\sigma$ for two actions (Exercise 10.11), $1.54\,\sigma$ for ten, $1.87\,\sigma$ for twenty (Figure 10.4a). Through bootstrapping, the bias of one backup becomes part of the target of the next, so it can accumulate over the effective horizon $1/(1-\gamma)$.

Double DQN. Double Q-learning (Algorithm 6.5) removes the bias by using two independent estimates, one to choose the maximizing action and the other to evaluate it. Hasselt et al. (2016) observed that DQN already has two networks, and that a nearly free version of the idea uses the online network to choose and the target network to evaluate:

$$y^{\text{Double}} = r + \gamma\,(1-\tau)\; Q_{\bar{w}}\Big(s',\; \arg\max_{a'} Q_w(s', a')\Big).$$

If the two networks' errors were independent, this estimate would be unbiased when all actions are equally good, and, when they differ, it would err toward underestimation, because it sometimes evaluates a suboptimal action (Figure 10.4a, dashed). Hasselt et al. (2016) showed on Atari

that DQN's value estimates were often far above the returns its policies achieved, that Double DQN's were much closer, and that its scores were better on many games.

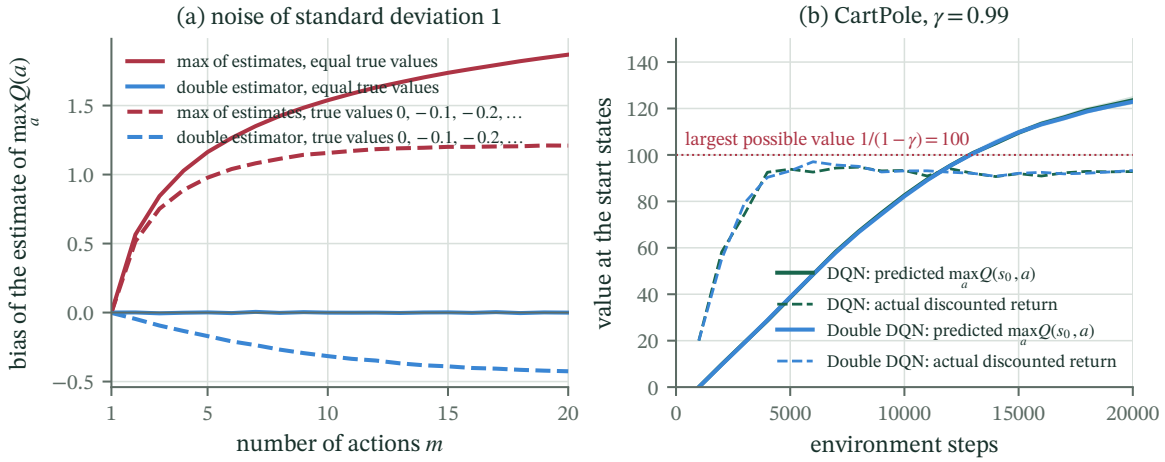


Figure 10.4. Overestimation. (a) The bias of two estimates of $\max_a Q(a)$ from estimates with independent standard normal errors, against the number of actions m: the maximum of the estimates (rose) and the double estimator, which picks the action with one set of estimates and evaluates it with another (blue); solid lines for equal true values, dashed for true values $0, -0.1, -0.2, \ldots$ (Monte Carlo with 200,000 samples). (b) CartPole: the value DQN and Double DQN predict at ten fixed start states (solid, mean of five seeds with ±1 standard error) and the discounted return their greedy policies actually obtain from those states (dashed). Both predictions pass 100, which no policy can reach, and keep rising; Double DQN barely changes this.

The CartPole experiment of Figure 10.4b is humbling. The predicted value of the start states rises steadily, passes the largest possible value 100 after about 13,000 steps, and is about 123 at 20,000 steps, while the greedy policies actually obtain about 93. Double DQN's predictions are almost the same. With only two actions the maximization bias of each backup is small, and with a target network only 100 updates old, the online and target networks make nearly the same errors, so the double estimator removes little. The overshoot also has a cause that no target formula addresses: a neural network fitted to targets need not stay below the largest target, as an averager would (Section 9.10). The lesson is not that Double DQN is useless (it is cheap and on large action spaces it helps), but that value estimates should be checked against returns, and that a single fix rarely removes a bias with several causes. The actor-critic methods of Chapter 13 use a stronger correction, the minimum of two independently trained critics (Algorithm 13.2).

CHECKPOINT

1. A transition has $r = 1$, $\gamma = 0.9$, a non-terminal s', target-network values $Q_{\bar{w}}(s', \cdot) = (1, 2)$ and online values $Q_w(s', \cdot) = (1.5, 1.2)$. What are the DQN and Double DQN targets?
2. Why is the replay buffer the source of the off-policy leg of the deadly triad in DQN?
3. With $C = 1000$ and values starting near 0, roughly how large can CartPole's values be after 20 target updates if every reward is 1 and $\gamma = 0.99$?

Answers.

1. DQN: $1 + 0.9 \times 2 = 2.8$. Double DQN: the online network prefers the first action, which the target network values at 1, so $1 + 0.9 \times 1 = 1.9$.
2. The buffer holds transitions generated by older, more exploratory policies, so the distribution of training data differs from that of the current greedy policy whose values the targets assume.
3. About $1 + \gamma + \cdots + \gamma^{20} = (1 - 0.99^{21})/0.01 \approx 19$: each target update lets the values grow by about one more discounted reward.

10.5 Dueling networks

In many states the choice of action hardly matters: when the pole is upright and centered, pushing left or right makes little difference to the long-run return, while the value of being in that state is high. A Q-network must nevertheless learn each action's value separately. The *dueling architecture* of Z. Wang et al. (2016) builds the distinction into the network. A shared torso computes features of the state; one head outputs a scalar $V(s)$ and another a vector $A(s, \cdot)$ of advantages, and they are combined into Q-values (Figure 10.5).

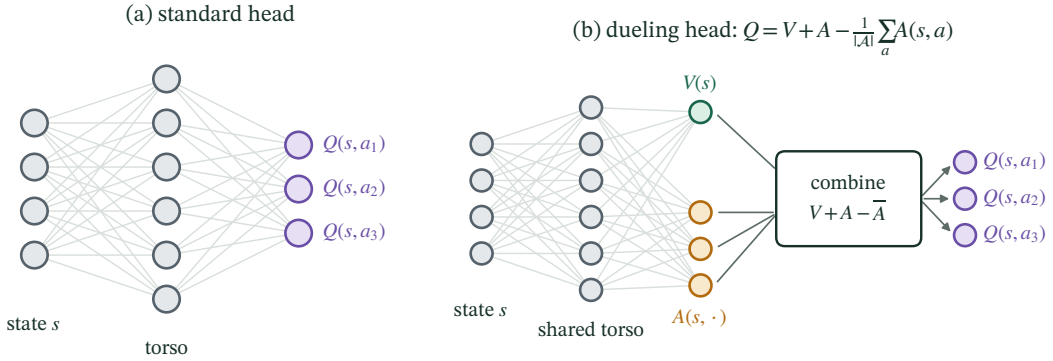


Figure 10.5. A standard Q-network head (a) and a dueling head (b). The dueling head computes a state value $V(s)$ and advantages $A(s, a)$ from shared features and combines them into $Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a')$. Every update changes $V(s)$, so what is learned about the state is shared by all actions.

The obvious combination $Q(s, a) = V(s) + A(s, a)$ has a flaw: it is *not identifiable*. Adding a constant c to $V(s)$ and subtracting it from every $A(s, a)$ gives the same Q-values, so the data cannot determine V and A separately, and the two heads can drift in opposite directions. The fix is to constrain the advantages. Subtracting their mean,

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a'),$$

makes the advantages average to zero, and then $V(s)$ is the mean of the Q-values over actions; subtracting their maximum instead makes $V(s) = \max_a Q(s, a)$, the value of the greedy policy (Exercise 10.4). Z. Wang et al. (2016) found the mean version more stable and used it. The gradient of the loss reaches $V(s)$ through every action's Q-value, so even updates of a single action improve the state value shared by all of them.

Dueling networks reported substantial gains on Atari (Z. Wang et al. 2016). On CartPole, with only two actions, our dueling agent learned no better and no worse than the standard one, which

is a reminder that architectural priors pay off where their assumption holds, here, many actions with similar values.

10.6 Prioritized experience replay

Uniform replay treats every stored transition as equally useful. But a transition whose TD error is already near zero has little to teach, while one with a large error, a surprise, may have a lot. *Prioritized experience replay* (Schaul et al. 2016) samples transitions with probability increasing in their last TD error, much as prioritized sweeping (Algorithm 8.3) orders planning updates. In the proportional variant, transition i has priority $p_i = |\delta_i| + \epsilon$, with a small $\epsilon > 0$ so that no transition is starved, and is sampled with probability

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha},$$

where $\alpha \in [0, 1]$ interpolates between uniform sampling ($\alpha = 0$) and fully proportional sampling ($\alpha = 1$). New transitions receive the largest priority seen so far, so that each is replayed at least once soon, and priorities are updated with the new TD errors whenever transitions are replayed.

Correcting the bias. Non-uniform sampling changes the distribution over which the loss is averaged, and therefore the solution: a transition that is sampled more often pulls the network toward its own target more. Prioritized replay corrects this with importance-sampling weights,

$$\omega_i = \left(\frac{1}{N\,P(i)}\right)^\beta, \qquad \text{normalized by } \max_j \omega_j,$$

which multiply each transition's loss. With $\beta = 1$ the weights exactly undo the non-uniform sampling in expectation, since $P(i)\,\omega_i \propto 1/N$; smaller β corrects only partly. Because the bias matters most near convergence, β is annealed from a smaller value toward 1 during training (values around $\alpha = 0.6$ and β from 0.4 to 1 are typical for the proportional variant (Schaul et al. 2016)). Normalizing by the largest weight only ever scales updates down, which keeps step sizes safe.

For example, with four transitions of priorities $1, 2, 3, 4$ and $\alpha = 1$, the sampling probabilities are $0.1, 0.2, 0.3, 0.4$; with $N = 4$ and $\beta = 1$, the weights $1/(NP(i))$ are $2.5, 1.25, 0.83, 0.63$, which normalize to $1, 0.5, 0.33, 0.25$. Without the correction, the fixed point of learning can move substantially (Exercise 10.12 gives a two-transition example in which it moves from 1.0 to 2.5).

The sum tree. Sampling from a distribution over a million transitions whose probabilities change after every minibatch needs a data structure. A *sum tree* stores the priorities in the leaves of a binary tree whose internal nodes hold the sums of their children. To sample, draw u uniformly in $[0, \text{total})$ and walk down from the root, going left if u is less than the left child's sum and otherwise subtracting that sum and going right; to update a priority, change its leaf and recompute the sums on the path to the root. Both cost $O(\log N)$. A minibatch is usually drawn by *stratified sampling*: the total is cut into $|B|$ equal segments and one u is drawn from each. The library's `SumTree` and `PrioritizedReplay` implement exactly this, vectorized over the minibatch.

Schaul et al. (2016) reported faster learning and better final scores than uniform replay on most Atari games, and prioritized replay became one of the most important components of Rainbow

(Section 10.7). It has costs and pitfalls. Large TD errors can come from noise rather than from information: in a stochastic environment, transitions with random rewards keep large errors forever and attract replay. Priorities are stale, computed when a transition was last replayed. And stronger prioritization makes training more off-policy, which increased soft divergence in the study of Hasselt et al. (2018). On CartPole our prioritized agent reached about the same returns as the uniform one, at roughly twice the computation per step.

10.7 Multi-step targets, noisy networks and Rainbow

Multi-step targets. DQN's one-step target propagates a reward back by one step per update of the target network, which is slow when rewards are delayed. The n-step return of (7.1) bootstraps after n steps instead:

$$y_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \left(1 - \tau_{t+n}\right) \max_{a'} Q_{\bar{w}}(s_{t+n}, a').$$

If the episode terminates within the n steps, the sum stops at the terminal transition and there is no bootstrap term; if it is truncated by a time limit after $k < n$ steps, the target bootstraps from the last state with factor γ^k. Rewards must never be summed across the end of an episode. The library's `NStep` class builds such transitions as experience arrives, and stores γ^k with each one. Used with replay, the intermediate actions come from an older behavior policy, so $y^{(n)}$ estimates a mixture of the greedy and the behavior policy's values, not Q^*; the off-policy corrections of Chapter 7 (for example Retrace, Section 7.8) fix this, but most agents simply use a small n such as 3 and accept the bias, because propagating rewards n steps per update, and putting only the weight γ^n on a bootstrapped estimate that is itself biased early in training, usually matter more. On CartPole, 3-step targets gave the largest improvement of any extension we tried: over the last 5000 of 20,000 steps, the average return was about 365 against 270 for one-step DQN (five seeds each).

Noisy networks. ε-greedy exploration is undirected: it dithers at random with the same probability in every state. *Noisy networks* (Fortunato et al. 2018) instead add learned noise to the weights of the network's linear layers, $y = (\mu^W + \sigma^W \odot \xi^W)\, x + \mu^b + \sigma^b \odot \xi^b$, where ξ is fresh random noise, $\odot$ is the elementwise product, and the means μ and the noise scales σ are learned by gradient descent along with everything else. Acting greedily with respect to a perturbed network gives exploration that depends on the state and can be reduced by the agent where it is not useful, as σ shrinks. To reduce the cost of sampling, the noise of a layer is usually *factorized* into an outer product of two noise vectors.

Rainbow. Hessel et al. (2018) combined six extensions of DQN in one agent: Double DQN, prioritized replay, the dueling architecture, 3-step targets, distributional learning with C51 (Section 10.8), and noisy networks. On the 57 Atari games, Rainbow outperformed every individual extension, both in final performance and in data efficiency. Its ablation study, which removed one component at a time, is as instructive as the agent: prioritized replay and multi-step targets were the two most important components, removing either caused the largest drops in median performance; distributional learning came next; noisy networks helped on some games and hurt on others; and removing Double DQN or the dueling head made little difference to the median,

although each mattered on particular games. The lesson is that improvements interact, and their value depends on the rest of the agent and on the benchmark. Later work showed that many of the same conclusions can be reached with far less computation on small environments (Obando-Ceron and Castro 2021), which makes such ablations accessible.

Our "Rainbow-lite" agent in the lab (Double DQN, a dueling head, 3-step targets and prioritized replay, without the distributional part and the noisy layers) reached returns around 400 on CartPole within 20,000 steps, the best of our variants.

10.8 Distributional reinforcement learning ●●

The value $Q^\pi(s,a)$ is the *mean* of the random return. On the Trail under the uniform policy, the return from Start is 1, 1.8 or 3.24, with probabilities $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{4}$; its mean is 1.76, a value the return never takes. *Distributional reinforcement learning* learns the whole distribution. Doing so for control was proposed by Bellemare et al. (2017), whose C51 agent improved DQN substantially on Atari, and it has since become a field of its own, with a textbook (Bellemare et al. 2023).

10.8.1 The distributional Bellman equation

Let $Z^\pi(s,a)$ denote the random return obtained by taking a in s and then following π, and $\eta^\pi(s,a)$ its distribution. The return satisfies the recursion $G_t = r_t + \gamma G_{t+1}$, so, with $s' \sim P(\cdot \mid s,a)$ and $a' \sim \pi(\cdot \mid s')$,

$$Z^\pi(s,a) \overset{D}{=} r(s,a) + \gamma\, Z^\pi(s',a'),$$

where $\overset{D}{=}$ means equality in distribution, the reward may itself be random, and $Z^\pi(s',a')$ is taken independent of the rest given s' and a' ($Z^\pi = 0$ at a terminal state). This is the *distributional Bellman equation*. The *distributional Bellman operator* $\mathcal{T}^\pi$ maps a collection of distributions η, one per state-action pair, to the distributions of the right-hand side: it shifts each next-state distribution by the reward, scales it by γ, and mixes over next states and actions. On the Trail, Goal's return is 4; Middle's is 2 (exit) or $0 + 0.9 \times 4 = 3.6$ (go), each with probability $\frac{1}{2}$; and Start's mixes 1 (exit) with 0.9 times Middle's distribution, giving 1, 1.8 and 3.24.

To measure how far apart two distributions are, the right tool is the *Wasserstein distance*. For distributions ν_1, ν_2 on the real line with cumulative distribution functions F_1, F_2, the p-Wasserstein distance is

$$W_p(\nu_1,\nu_2) = \Big(\int_0^1 |F_1^{-1}(u) - F_2^{-1}(u)|^p \, du \Big)^{1/p},$$

the average distance between matching quantiles. Its maximal form over state-action pairs is $\bar{d}_p(\eta_1,\eta_2) = \max_{s,a} W_p(\eta_1(s,a),\eta_2(s,a))$.

Theorem 10.1 The distributional Bellman operator is a contraction (Bellemare, Dabney and Munos 2017)

For a finite MDP, a policy π, $\gamma < 1$ and any $p \in [1,\infty]$, the distributional Bellman operator $\mathcal{T}^\pi$ is a γ-contraction in $\bar{d}_p$, on the set of return distributions with finite p-th moments. Consequently, repeated application of $\mathcal{T}^\pi$ converges to the return distribution η^π from any starting point.

The intuition (Exercise 10.9): scaling two random variables by γ scales the distance between their quantiles by γ; adding the same reward to both moves them together; and mixing pairs of distributions with the same weights does not increase the distance beyond the largest pairwise distance. The statement fails in other metrics, for example the total variation distance, which is not reduced by scaling. In the control setting, the *distributional optimality operator*, which uses the greedy action of the next state's mean, need not be a contraction in any of these metrics, although the means of its iterates still converge to Q^* as in ordinary value iteration (Bellemare et al. 2017). Control is therefore more delicate in theory than evaluation.

10.8.2 Categorical DQN (C51)

A distribution must be represented with finitely many numbers. C51 (Bellemare et al. 2017) fixes N evenly spaced *atoms* $z_1 < z_2 < \cdots < z_N$ with spacing Δz (on Atari, 51 atoms from -10 to 10, which gave the method its name) and lets the network output, for each action, a probability vector over the atoms through a softmax: $Z_w(s,a)$ takes the value z_j with probability $p_j(s,a)$, and $Q_w(s,a) = \sum_j z_j\, p_j(s,a)$.

The target distribution $r + \gamma Z(s', a^*)$ puts probability $p_j(s', a^*)$ on the point $r + \gamma z_j$, which in general is not an atom. C51 *projects* it back: each shifted point is clipped to $[z_1, z_N]$, and its probability is split between the two neighboring atoms in proportion to closeness, so that a point a quarter of the way from z_i to z_{i+1} gives three quarters of its mass to z_i and one quarter to z_{i+1}. The projection preserves total probability, and, unless clipping occurs, it preserves the mean. The network is then trained to minimize the cross-entropy between the projected target m and its prediction, $-\sum_j m_j \log p_j(s,a)$, whose gradient with respect to the logits is simply $p - m$.

Algorithm 10.2 The categorical (C51) target

Input: a transition (s, a, r, s', τ), atoms $z_1, \dots, z_N$ with spacing Δz, target network probabilities $p_j(s', \cdot)$, discount γ.

1. $a^* \leftarrow \arg\max_{a'} \sum_j z_j\, p_j(s', a')$ (the greedy action of the target means).
2. $m \leftarrow 0 \in \mathbb{R}^N$.
3. For $j = 1, \dots, N$:
   1. $\hat{z} \leftarrow \mathrm{clip}\big(r + \gamma(1-\tau) z_j,\ z_1,\ z_N\big)$; $b \leftarrow (\hat{z} - z_1)/\Delta z$; $l \leftarrow \lfloor b \rfloor$, $u \leftarrow \lceil b \rceil$ (atoms are indexed from 0 here).
   2. If $l = u$: $m_l \leftarrow m_l + p_j(s', a^*)$; else: $m_l \leftarrow m_l + p_j(s', a^*)\,(u - b)$ and $m_u \leftarrow m_u + p_j(s', a^*)\,(b - l)$.

Output: the projected target m; the loss is $-\sum_i m_i \log p_i(s,a)$.

Example 10.1 A categorical backup on the Trail

Use the atoms $0, 1, 2, 3, 4$ and the uniform policy with $\gamma = 0.9$. Goal's return, 4, is an atom. From Middle, exiting returns 2, an atom; going on returns $0.9 \times 4 = 3.6$, which the projection splits

into 0.4 on 3 and 0.6 on 4. So the categorical distribution of Middle is 0.5 on 2, 0.2 on 3 and 0.3 on 4, with mean $1 + 0.6 + 1.2 = 2.8 = V^{\pi}(\text{Middle})$ (Figure 10.6a).

Now back it up to (Start, go): the reward is 0 and the atoms scale to 1.8, 2.7 and 3.6 (Figure 10.6b). The projection sends 1.8 to 1 and 2 with weights 0.2 and 0.8; 2.7 to 2 and 3 with weights 0.3 and 0.7; and 3.6 to 3 and 4 with weights 0.4 and 0.6. Multiplying by the probabilities 0.5, 0.2, 0.3 and adding,

$$m = (0,\ 0.10,\ 0.46,\ 0.26,\ 0.18),$$

whose mean is $0.10 + 0.92 + 0.78 + 0.72 = 2.52 = 0.9 \times 2.8 = Q^{\pi}(\text{Start}, \text{go})$, as promised (Figure 10.6c). Mixing with the exit's point mass at 1 gives the categorical distribution of Start: $(0, 0.55, 0.23, 0.13, 0.09)$, with mean 1.76.

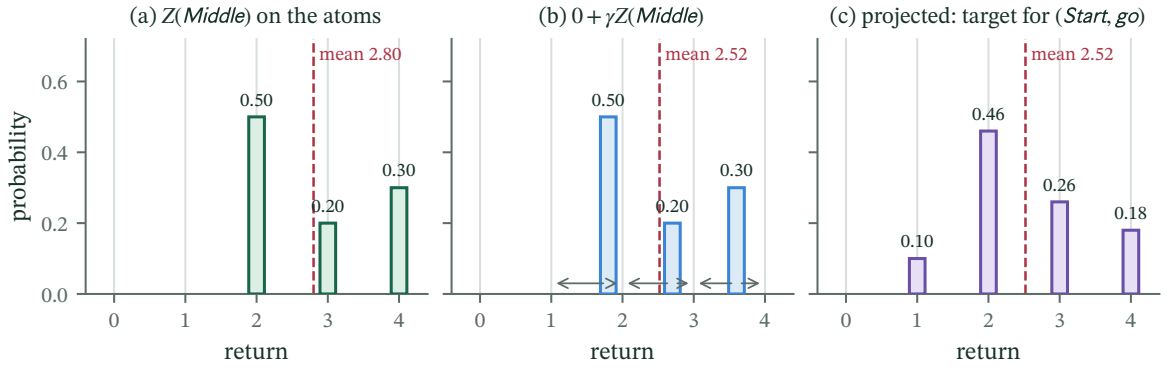


Figure 10.6. One categorical backup on the Trail (uniform policy, $\gamma = 0.9$, atoms 0 to 4). (a) The categorical return distribution of Middle. (b) Shifted by the reward 0 of go and scaled by γ, its atoms land at 1.8, 2.7 and 3.6, between the grid points; the arrows show where their probability goes. (c) After the projection: the target distribution for (Start, go). The dashed lines mark the means, which the projection preserves.

Notice that the categorical distribution of Start obtained by repeated projected backups, $(0.55, 0.23, 0.13, 0.09)$ on atoms 1 to 4, is not the projection of the true distribution onto the atoms, which is $(0.55, 0.20, 0.19, 0.06)$: projection errors made at Middle are backed up into Start (the lab's categorical TD converges to the former). Both have the correct mean. Rowland et al. (2018) analyzed this projected operator: the C51 projection is an orthogonal projection in the geometry of the *Cramér distance* $\left(\int (F_1 - F_2)^2\right)^{1/2}$, the projected operator is a $\sqrt{\gamma}$-contraction in its maximal form, and the error of its fixed point is controlled by the spacing of the atoms.

10.8.3 Quantile regression: QR-DQN and IQN

C51 needs the range $[z_1, z_N]$ in advance, and its loss is not a distance between distributions. *QR-DQN* (Dabney, Rowland, et al. 2018) turns the representation around: it fixes N probabilities, each $\frac{1}{N}$, and learns the locations. The network outputs, for each action, N numbers $\theta_1, \dots, \theta_N$, meant to be the quantiles of the return at the levels $\hat{\tau}_i = \frac{2i-1}{2N}$, the midpoints of N equal slices of probability. For $N = 4$ these are $\frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}$. The projection onto such distributions that minimizes

the 1-Wasserstein distance is exactly "take the quantiles at the midpoints", and Dabney, Rowland, et al. (2018) showed that this projection composed with $\mathcal{T}^\pi$ is a γ-contraction in $\bar{d}_\infty$.

Quantiles can be learned by *quantile regression* (Koenker and Bassett 1978). For a level $\tau \in (0,1)$, the *quantile loss* of a prediction θ and an outcome z is

$$\rho_\tau(z-\theta) = (z-\theta)\big(\tau - \mathbb{1}\{z < \theta\}\big),$$

an asymmetric absolute value that charges underestimates with weight τ and overestimates with weight $1-\tau$ (Figure 10.7a). Its expectation $\mathbb{E}[\rho_\tau(Z-\theta)]$ is minimized exactly by the τ-quantiles of Z (Exercise 10.10), and its gradient with respect to θ is $-(\tau - \mathbb{1}\{z<\theta\})$: push θ up with force τ when the outcome lies above it and down with force $1-\tau$ when below, which balances when a fraction τ of outcomes lies below. QR-DQN's targets are the N points $r + \gamma\,\theta_j(s', a^*)$, and each $\theta_i(s,a)$ is regressed toward all of them with the loss at level $\hat{\tau}_i$. In practice the kink at zero is smoothed with a Huber loss, $|\tau - \mathbb{1}\{u<0\}|\,\ell_\kappa(u)/\kappa$, the *quantile Huber loss*.

Example 10.2 Quantiles of the Trail's return

With $N = 4$, the true quantiles of Start's return at levels $\frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}$ are $1, 1, 1.8, 3.24$: half the probability sits at 1, so the first two levels fall there, the next lies in the quarter at 1.8, and the last in the quarter at 3.24. The four quantiles reproduce the distribution exactly, and their average is 1.76. Tabular quantile TD, the QR-DQN update without a network, converges to these values on the Trail; the lab finds 0.99, 1.01, 1.80, 3.24. Figure 10.7b shows the version with eight quantiles. Quantiles at levels where the distribution function is flat are not unique: here $F = \frac{1}{2}$ on $[1, 1.8)$, so every value between 1 and 1.8 is a $\frac{1}{2}$-quantile.

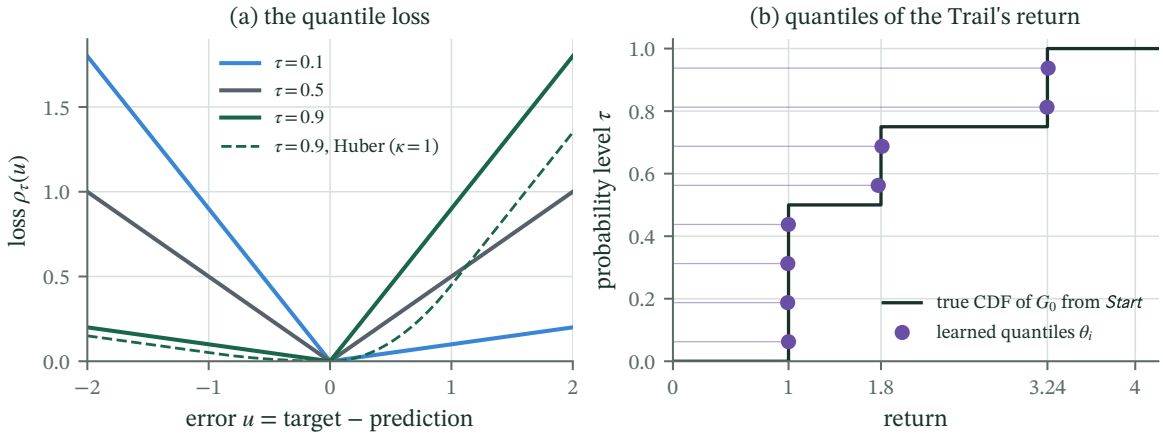


Figure 10.7. Quantile regression. (a) The quantile loss $\rho_\tau(u)$ against the error $u = z - \theta$: an asymmetric absolute value; the dashed curve is its Huber version for $\tau = 0.9$, quadratic near zero. (b) The distribution function of the return from Start on the Trail (uniform policy, $\gamma = 0.9$) and the eight quantiles θ_i learned by tabular quantile TD at the levels $\tau_i = (2i-1)/16$, drawn at height τ_i: they sit on the steps of the true distribution.

Implicit quantile networks (IQN) (Dabney, Ostrovski, et al. 2018) go one step further: the network takes the level τ as an extra input and outputs the corresponding quantile, so it represents the whole quantile function and can be queried at any τ. A policy can then maximize a risk-sensitive criterion, for example the mean of the lower quantiles, by changing the levels at which it averages, with no change to training.

10.8.4 Why does it help?

A greedy policy needs only means, and the distributional Bellman update of a mean is the ordinary Bellman update. So why does learning the whole distribution improve performance, as it did for C51, QR-DQN and IQN on Atari, and in Rainbow's ablation? The answer is still not fully settled, but it is clear that the benefit comes from how the distribution interacts with function approximation, not from the expected values themselves. Lyle et al. (2019) showed that, in the tabular setting and in several linear settings, distributional updates produce exactly the same expected values as their expected counterparts; differences arise with nonlinear networks. The leading explanations are about representation: predicting many statistics of the return is a rich auxiliary task that shapes the shared features, and the cross-entropy or quantile losses behave better as regression losses than the squared error, with better-conditioned gradients when targets change scale. Learned distributions also have uses of their own, in risk-sensitive control and in exploration, which Bellemare et al. (2023) survey.

10.9 Scaling up ●●●

DQN's success raised two opposite questions. How far can value-based agents go with more computation? And how little data do they really need?

Distributed agents. *Ape-X* (Horgan et al. 2018) separated acting from learning: hundreds of actors, each with its own exploration rate, generate experience in parallel and write it into a shared prioritized replay buffer, computing the initial priorities themselves; a single learner trains on it. With far more data per hour, Ape-X reached much higher Atari scores in a fraction of the wall-clock time. *R2D2* (Kapturowski et al. 2019) added a recurrent network (an LSTM, Section 19.5.2) and replayed sequences instead of transitions. It addressed the problem that the recurrent state stored with old sequences is stale by replaying a short "burn-in" prefix of each sequence to warm the state up before learning on the rest, and it reported large gains on Atari. *Agent57* (Badia, Piot, et al. 2020) built on R2D2 and on the intrinsic-motivation agent *Never Give Up* (Badia, Sprechmann, et al. 2020) (Section 16.7), with an adaptive mechanism that chooses among exploration strategies and discount factors during training, and was the first agent to exceed the human benchmark on all 57 Atari games, including the hardest exploration games. These agents used billions of frames.

Data efficiency. At the other extreme, the *Atari 100k* benchmark allows only 100,000 agent steps per game, about two hours of play (Kaiser et al. 2020). Simply retuning Rainbow for this regime, with more updates per step and multi-step targets, already gave a strong baseline (Hasselt et al. 2019). Self-predictive representations (Schwarzer et al. 2021) added an auxiliary loss that predicts future latent states, and periodic *resets* of the network allowed much higher replay ratios

(D'Oro et al. 2023). *BBF* (Schwarzer et al. 2023) combined a much larger network, a high replay ratio, resets, self-prediction, and schedules that increase the discount and shorten the multi-step horizon during training, and reported human-level aggregate performance on Atari 100k.

Plasticity. A thread runs through these results: deep value-based agents lose the ability to learn as training continues. Trained on early, limited data, a network can overfit in ways it never recovers from, the *primacy bias* (Nikishin et al. 2022); networks trained on changing targets gradually lose capacity to fit new ones (Lyle et al. 2022); many units stop responding to any input, becoming "dormant" (Sokar et al. 2023); and bootstrapping can collapse the rank of the learned features (Kumar et al. 2021). Resetting part of the network, or perturbing its weights, restores plasticity, and is part of why high replay ratios became usable. How much the replay ratio, the network size and the reset schedule can be pushed, and why larger networks help value-based agents only when combined with such measures, are active research questions; Section 24.5 returns to plasticity in the setting of continual learning.

RESEARCH NOTE

Open questions: Why does training on bootstrapped targets degrade a network's plasticity more than supervised learning on fixed targets? Is there a principled replacement for periodic resets? How should the replay ratio, network size and amount of data scale together, in the way that scaling laws relate them in supervised learning? Exercises 10.16 and 10.17 are starting points.

10.10 Practice

Hyperparameters that matter. In our CartPole search, and consistently in the literature, a few choices matter much more than the rest. The *step size* and optimizer: Adam with η around 10^{-4} to 10^{-3} is the usual starting point. The *target update period* C (or Polyak rate τ): too large and values propagate slowly (with $C = 500$, our CartPole values were still near 35 after 20,000 steps), too small and the estimates overshoot or diverge. The *replay buffer size and replay ratio*: a larger buffer is more diverse but more off-policy, and Fedus et al. (2020) found that the benefit of larger buffers depends strongly on using multi-step targets. The *multi-step horizon* n, often 3. The *exploration schedule*: decaying ε too fast commits the agent before its values mean anything. The *loss and the reward scale* together: the Huber loss with clipped rewards, or the squared loss with rewards and values of moderate size. The *network size*: on CartPole, one hidden layer of 128 units learned faster and more reliably than two layers of 64. Tune these first, with several seeds, and change one thing at a time; Section 23.5 discusses tuning in general.

A debugging checklist. When a DQN does not learn, the cause is usually a bug rather than a hyperparameter. Check, in roughly this order:

- *Targets:* terminal transitions do not bootstrap; truncated ones do; the target is computed with the target network and blocks gradients; n-step sums stop at episode boundaries.
- *Indexing:* the loss uses the Q-value of the action actually taken, with the right shape and broadcast.
- *Magnitudes:* predicted values stay within the range the rewards allow, roughly $[r_{\min}, r_{\max}]/(1-\gamma)$; values far outside it signal divergence or a target bug.

- *Data:* the buffer stores what you think it stores (print a few transitions); sampling is uniform (or as prioritized); learning starts only after the buffer holds enough data.
- *Exploration:* ε follows its schedule; the greedy action is computed from the online network.
- *Evaluation:* the greedy policy is evaluated separately from the training episodes, with a small or zero ε, over several episodes and seeds.

Logging the mean predicted value, the TD errors and the gradient norm alongside the returns catches most problems early; Section 23.3 gives a fuller procedure.

Evaluation protocols. Atari results are only comparable if they are evaluated the same way, and early protocols differed. Machado et al. (2018) proposed a standard: *sticky actions*, under which the environment repeats the agent's previous action with probability 0.25, to make the deterministic emulator stochastic so that agents cannot memorize action sequences; evaluation at fixed points during training, averaged over many episodes; and many independent runs. Earlier work had injected randomness with up to 30 "no-op" actions at the start of each episode (Mnih et al. 2015), which still leaves the rest of the episode deterministic. Scores are usually normalized by those of a human and a random player and aggregated over games; R. Agarwal et al. (2021) showed that with few runs the median of normalized scores is unreliable and recommended the interquartile mean with bootstrap confidence intervals. Section 23.4 treats evaluation in general.

10.11 Summary

- DQN is Q-learning with a neural network, made stable enough to work by two devices that make it resemble fitted Q iteration: a replay buffer, which provides decorrelated and reusable data, and a target network, which holds the regression targets fixed for C steps.
- DQN still has all three legs of the deadly triad. On CartPole, removing the target network made the value estimates drift far beyond any possible value and the returns collapse after an early peak; removing replay slowed learning.
- The target period controls how fast values propagate: roughly one discounted reward per target update.
- Q-learning overestimates because it maximizes over noisy estimates; Double DQN evaluates the online network's choice with the target network. On CartPole both overestimated, and the double estimator helped little; on Atari it helped substantially.
- Dueling networks separate the state value from centered action advantages; prioritized replay samples surprising transitions more often and corrects the bias with importance weights; multi-step targets propagate rewards faster; noisy networks learn their exploration. Rainbow combines them all, and its ablation found prioritization and multi-step targets the most valuable.
- (●●) Distributional RL learns the distribution of the return. The distributional Bellman operator is a γ-contraction in the maximal Wasserstein distance; C51 represents distributions on fixed atoms with a projection that preserves the mean, and QR-DQN learns quantiles by quantile regression. The benefits come through representation learning, not through the means themselves.
- (●●●) Distributed agents (Ape-X, R2D2, Agent57) and data-efficient ones (SPR, BBF) pushed value-based methods to both extremes, and revealed the loss of plasticity as a

central problem.
- In practice: tune the step size, the target period, the replay ratio, n and the exploration schedule first; check the targets before anything else; and evaluate with sticky actions, many seeds and robust aggregates.

The next part of the book changes perspective: instead of learning values and acting greedily, the methods of Chapter 11 optimize the policy directly.

10.12 Further Reading

The two DQN papers are short and readable: Mnih et al. (2013) for the original idea and Mnih et al. (2015) for the full agent, its preprocessing and its ablations. Lin (1992) introduced experience replay and is worth reading for how many later ideas it anticipates. The extensions are best read in their original papers: Double DQN (Hasselt et al. 2016), dueling networks (Z. Wang et al. 2016), prioritized replay (Schaul et al. 2016), noisy networks (Fortunato et al. 2018), and their combination in Rainbow (Hessel et al. 2018), whose ablation figure summarizes a lot of experience in one picture. Obando-Ceron and Castro (2021) repeat many of these studies at small scale. For the stability of deep Q-learning, Hasselt et al. (2018) is the most systematic empirical study, and Fedus et al. (2020) analyze the role of the replay buffer.

(●●) Distributional reinforcement learning has a textbook: Bellemare et al. (2023) covers the theory and the algorithms in depth. The original papers are Bellemare et al. (2017) (C51 and the contraction results), Dabney, Rowland, et al. (2018) (QR-DQN) and Dabney, Ostrovski, et al. (2018) (IQN); Rowland et al. (2018) analyze the categorical algorithm, and Lyle et al. (2019) ask when distributional learning can make a difference. Koenker and Bassett (1978) is the classic reference on quantile regression.

(●●●) For scaling, read Ape-X (Horgan et al. 2018), R2D2 (Kapturowski et al. 2019) and Agent57 (Badia, Piot, et al. 2020). For data efficiency, start with Hasselt et al. (2019) and follow the line through Schwarzer et al. (2021), D'Oro et al. (2023) and Schwarzer et al. (2023). For plasticity, see Nikishin et al. (2022), Lyle et al. (2022) and Sokar et al. (2023). For evaluation, Machado et al. (2018) and R. Agarwal et al. (2021) are essential.

10.13 Exercises

Exercise 10.1 ● How replay decorrelates

A replay buffer holds 50,000 transitions from 250 episodes of 200 steps each. (a) Two transitions are drawn independently and uniformly from the buffer. What is the probability that they come from the same episode? (b) In a minibatch of 64, how many of the $\binom{64}{2}$ pairs come from the same episode, on average? Compare with a minibatch of 64 consecutive transitions. (c) Give two reasons why correlated minibatches are harmful for training a neural network, and one reason why replay cannot remove all correlation.

Exercise 10.2 ●● The target network as delayed fitted Q iteration

Consider DQN on CartPole (reward 1 per step, $\gamma = 0.99$) with learning starting after 1000 steps, one gradient step per environment step, and a target network copied every C gradient steps. (a) Assume each target period solves its regression exactly, the network starts at $Q = 0$, and the regression never predicts more than the largest target. Show that after K target copies every value is at most $(1 - \gamma^{K+1})/(1 - \gamma)$. (b) Evaluate the bound after 20,000 steps for $C = 500$ and for $C = 100$. (c) Our runs gave values near 35 for $C = 500$ and near 123 for $C = 100$. Which assumption of (a) explains each observation?

Exercise 10.3 ● Double DQN targets by hand

For three transitions with rewards $(1, 1, 0)$, $\gamma = 0.9$, the third one terminal, target-network values $Q_{\bar{w}}(s', \cdot) = (1, 2), (3, 0.5), (2, 2)$ and online values $Q_w(s', \cdot) = (1.5, 1.2), (2, 2.5), (0, 1)$: (a) compute the DQN targets; (b) compute the Double DQN targets; (c) explain why the Double DQN targets are never larger than the DQN targets with the same networks.

Exercise 10.4 ● Identifiability of the dueling head

(a) Show that $Q(s, a) = V(s) + A(s, a)$ does not determine V and A from Q. (b) With the mean-subtracted combination of Section 10.5, show that $V(s) = \frac{1}{|\mathcal{A}|} \sum_a Q(s, a)$, and find A from Q. (c) With the max-subtracted combination $Q = V + A - \max_{a'} A(s, a')$, show that $V(s) = \max_a Q(s, a)$. (d) A dueling head outputs $V(s) = 2$ and $A(s, \cdot) = (0.5, -0.5, 1)$. What Q-values does each combination produce, and does the greedy action differ?

Exercise 10.5 ● Prioritized replay by hand

A buffer holds four transitions with priorities $p = (1, 2, 3, 4)$. (a) Compute the sampling probabilities for $\alpha = 0.6$. (b) Compute the importance-sampling weights for $\beta = 0.4$, normalized by their maximum. (c) Repeat for $\alpha = 1, \beta = 1$, and check that $P(i)\,\omega_i$ is the same for every i before normalization. (d) Why is the priority of a new transition set to the largest priority so far?

Exercise 10.6 ● Multi-step targets at episode boundaries

An episode produces the rewards $1, 0, 2, 4$ from states $s_0, \ldots, s_3$ and terminates after the fourth step; $\gamma = 0.9$ and $n = 3$. (a) Write the 3-step target of each of the four transitions, saying which ones bootstrap and from which state. (b) Repeat when the episode is not terminated but truncated by a time limit after the fourth step. (c) What goes wrong if a buffer of consecutive transitions is used to form n-step sums without checking for episode ends?

Exercise 10.7 ● A coarse categorical distribution

Repeat Example 10.1 with only three atoms, 0, 2 and 4. (a) Compute the categorical distribution of Middle and its mean. (b) Compute the projected target for (Start, go) and the categorical

distribution of Start. (c) Check that the means are 2.8, 2.52 and 1.76. What is odd about the probability assigned to a return of 0?

Exercise 10.8 ●● Huber loss and error clipping

(a) Derive the gradient of the Huber loss $\ell_\kappa(y - Q_w(s,a))$ with respect to w and show that it equals the gradient of the squared loss with the error clipped to $[-\kappa, \kappa]$. (b) Show that minimizing $\mathbb{E}[\ell_\kappa(Y - q)]$ over a constant q gives the mean of Y when $|Y - q| \le \kappa$ always holds, and something between the mean and the median otherwise. (c) On CartPole the values approach 100 and early TD errors are large. Explain why the Huber loss with $\kappa = 1$ can learn more slowly than the squared loss there, even with Adam, and what DQN's reward clipping has to do with the choice.

Exercise 10.9 ●● Why the distributional operator contracts

Work with the 1-Wasserstein distance W_1 between distributions on the real line. (a) Show that $W_1(\mathrm{law}(c + \gamma X), \mathrm{law}(c + \gamma Y)) = \gamma\, W_1(\mathrm{law}(X), \mathrm{law}(Y))$ for constants c and $\gamma \ge 0$, using the quantile form of W_1. (b) Show that for mixtures with the same weights, $W_1(\sum_i q_i \nu_i, \sum_i q_i \nu'_i) \le \sum_i q_i W_1(\nu_i, \nu'_i)$. (Hint: $W_1(\nu, \nu') = \int |F_\nu(x) - F_{\nu'}(x)|\, dx$.) (c) Combine (a) and (b) to prove Theorem 10.1 for $p = 1$ and deterministic rewards. (d) Why does the same argument not show that $\mathcal{T}^\pi$ contracts in total variation?

Exercise 10.10 ●● The quantile loss is minimized by the quantile

Let Z be a random variable with distribution function F and finite mean, and $\tau \in (0,1)$. (a) Show that $g(\theta) = \mathbb{E}[\rho_\tau(Z - \theta)]$ is convex, with right derivative $F(\theta) - \tau$ and left derivative $F(\theta^-) - \tau$. (b) Conclude that θ minimizes g if and only if $F(\theta^-) \le \tau \le F(\theta)$, that is, θ is a τ-quantile. (c) For the return from Start on the Trail, find the set of minimizers for $\tau = \frac{3}{8}$, $\tau = \frac{1}{2}$ and $\tau = \frac{7}{8}$.

Exercise 10.11 ●● The bias of the maximum

(a) Let X_1, X_2 be independent $N(0, \sigma^2)$. Show that $\mathbb{E}[\max(X_1, X_2)] = \sigma/\sqrt{\pi}$. (b) Let $\hat{Q}^A$ and $\hat{Q}^B$ be independent unbiased estimates of the values $\mu_1, \dots, \mu_m$. Show that the double estimator $\hat{Q}^B(\arg\max_a \hat{Q}^A(a))$ has expectation at most $\max_a \mu_a$, with equality when all μ_a are equal. (c) Explain why Double DQN's two estimates are not independent, and how the target period C affects their dependence.

Exercise 10.12 ●● Prioritized replay changes the answer

A state s has a single action, and the replay buffer contains its transitions with two possible targets: $y = 0$ in 90% of the stored transitions and $y = 10$ in 10%. A tabular estimate Q is trained on minibatches from the buffer. (a) What is the fixed point with uniform replay? (b) With proportional prioritization, $\alpha = 1$, priorities $|y - Q|$ and no importance weights, find the

fixed point of the expected update. (c) Show that with $\beta = 1$ the importance-sampling weights restore the uniform answer. (d) What happens with $0 < \beta < 1$?

Exercise 10.13 ●● The categorical fixed point on the Trail

(a) Show that the categorical TD fixed point for Start on the atoms $0, \dots, 4$ (uniform policy, $\gamma = 0.9$) is $(0, 0.55, 0.23, 0.13, 0.09)$, and that the projection of the true distribution is $(0, 0.55, 0.20, 0.19, 0.06)$. (b) Compute the 1-Wasserstein distance from the true distribution to each. (c) Compute the 1-Wasserstein distance from the true distribution to the four-quantile representation $(1, 1, 1.8, 3.24)$. What does the comparison suggest about the two representations on this problem?

Exercise 10.14 ●● Backpropagation through the dueling head

Let $Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a')$ and let $g_a = \partial L/\partial Q(s,a)$. (a) Show that $\partial L/\partial V(s) = \sum_a g_a$ and $\partial L/\partial A(s,a) = g_a - \frac{1}{|\mathcal{A}|}\sum_{a'} g_{a'}$. (b) In DQN only the action taken receives a nonzero g_a. What fraction of that gradient reaches V, and what happens to the advantages of the other actions? (c) Check your answer against `QNetwork.backward` in `rlcourse/dqn.py`.

Exercise 10.15 ● Ablations with more seeds (programming)

Extend Task 4 of Lab 10.1. (a) Run full DQN, no target network, no replay, and neither, with eight seeds each for 25,000 steps, and plot the mean returns with error bands and the predicted start values. (b) Which differences are larger than the variation between seeds? (c) Interpret the "neither" variant in terms of the deadly triad: which leg does training only on recent transitions weaken?

Exercise 10.16 ●●● Plasticity and the replay ratio

Write a short research proposal on the interaction between the replay ratio and the loss of plasticity in value-based agents. Summarize what Nikishin et al. (2022), D'Oro et al. (2023) and Lyle et al. (2022) found, state one hypothesis about the mechanism, and design an experiment with the lab's CartPole DQN (for example: replay ratios of 1, 4 and 16 updates per step, with and without periodic resets of the output layer) that would test it. Say what you would measure besides the return.

Exercise 10.17 ●●● The deadly triad in deep Q-learning

Read Hasselt et al. (2018). (a) Summarize their definition of soft divergence and their main findings about target networks, multi-step returns, Double DQN and prioritization. (b) Using the lab code, design and run a small version of one of their experiments on CartPole, measuring how often the predicted start values exceed 100, and report whether your results agree.

Exercise 10.18 ●●● Why does distributional learning help?

Explain the result of Lyle et al. (2019) that in the tabular setting distributional and expected updates produce the same expected values. Then list three hypotheses for why distributional agents nevertheless outperform DQN on Atari, and for each, an experiment that could support or refute it.

Part IV

Policy Optimization

Instead of learning values and acting greedily, an agent can adjust its policy directly in the direction of higher return. Part IV develops this idea from the policy gradient theorem (Chapter 11) through trust regions and PPO (Chapter 12) to the off-policy actor-critic methods used for continuous control, such as SAC and TD3 (Chapter 13). Chapter 14 explains why these methods find good policies despite optimizing a non-convex objective.

CHAPTER 11

Policy Gradient Methods

WHAT YOU WILL LEARN

- Explain why it can pay to optimize a parameterized policy directly instead of acting greedily on a learned value function.
- Compute the gradient of the Trail's expected return with respect to its logit parameters by hand, and read each factor of the result.
- State the policy gradient theorem and use the score-function trick to derive the REINFORCE estimator.
- Explain why a baseline reduces variance without adding bias, and measure the effect exactly on the Trail.
- Implement REINFORCE, one-step actor-critic, A2C and generalized advantage estimation, with the correct treatment of terminated and truncated episodes.
- Prove the policy gradient theorem in two ways and derive the variance-minimizing baseline (●●).
- Explain what implementations lose when they drop the γ^t factor, and state the compatible function approximation theorem (●●●).

Every control method in this book so far has learned values and derived its behavior from them. Q-learning estimates Q^* and acts greedily; SARSA and DQN do the same with an exploration rule bolted on. The policy is never represented on its own: it is whatever the value function says is best. This works remarkably well, but it ties the quality of the behavior to the quality of a maximization over actions, and that maximization is awkward when actions are continuous, brittle when the value function is approximate, and unable to represent behavior that should stay random.

This chapter takes the other road. We give the policy its own parameters θ, write the expected return $J(\theta)$ as a function of them, and climb it by gradient ascent. You have seen the idea once already: the gradient bandit of Section 2.6 adjusted softmax preferences in the direction that increases expected reward. The new difficulty is time. In a sequential problem an action changes not only the reward it earns but also every state that follows, and the gradient must account for all of it. The *policy gradient theorem* shows that it can, with a formula that needs no model of the environment and that can be estimated from sampled episodes.

From that theorem grows a family of methods that dominate modern practice. REINFORCE estimates the gradient with Monte Carlo returns. Baselines and learned critics reduce its variance, turning it into *actor-critic* methods. Generalized advantage estimation blends Monte Carlo and temporal-difference targets with one parameter. The trust-region methods of Chapter 12, includ-

ing PPO, are built directly on top of this chapter, and so is much of reinforcement learning for language models (Chapter 22).

The roadmap is as follows. Section 11.1 explains when and why direct policy optimization helps. Section 11.2 introduces softmax and Gaussian policies and computes the Trail's gradient by hand. Section 11.3 states the policy gradient theorem, with two proofs in ●● subsections. Section 11.4 to 11.7 build the algorithms: REINFORCE, baselines, actor-critic and A2C, and generalized advantage estimation. Section 11.8 collects the practical details that make these methods work. Section 11.9 (●●) examines the discount factor that most implementations quietly drop, average-reward gradients and reparameterization gradients, and Section 11.10 (●●●) covers compatible function approximation and the convergence of actor-critic methods. On a first reading, the ● sections form a complete path from the idea to working CartPole agents.

11.1 Why optimize the policy directly

A value-based agent chooses actions by solving a small optimization problem in every state: find the action with the largest estimated value. Four situations make this indirect route unattractive.

Continuous actions. If the action is a torque between -2 and 2, or a vector of twenty joint torques, then "the action with the largest Q" is a continuous optimization problem that has to be solved at every step. Discretizing the action space works for one or two dimensions and fails beyond that. A policy that outputs the action, or a distribution over actions, sidesteps the maximization entirely.

Behavior that should stay random. Some problems have no good deterministic policy. In rock–paper–scissors against an adaptive opponent, any deterministic choice can be exploited; the best policy plays each move with probability $\frac{1}{3}$ (Chapter 21). When two states look identical to the agent but require different actions, a policy that randomizes can do better than any deterministic one (Chapter 19). A greedy policy is deterministic by construction; a parameterized stochastic policy can represent these solutions directly.

Smooth changes. A greedy policy can change abruptly. If $Q(s, \text{left})$ and $Q(s, \text{right})$ are close, a tiny change in the estimates flips the action in s, and with function approximation one update can flip it in many states at once. A parameterized policy changes its action probabilities a little at a time, which makes learning more stable and easier to analyze.

Simple policies, complicated values. Sometimes the policy is much simpler than the value function. To balance a pole, "push the cart toward the side the pole is falling" is nearly enough, while the value of every state depends on the exact dynamics. Learning the simple object directly can be easier than learning the complicated one.

The recipe is short. Choose a family of policies $\pi_\theta(a \mid s)$ with parameters $\theta \in \mathbb{R}^d$, differentiable in θ. Measure performance by the *objective*

$$J(\theta) = \mathbb{E}_{s_0 \sim d_0}\big[V^{\pi_\theta}(s_0)\big] = \mathbb{E}_{\pi_\theta}\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big], \tag{11.1}$$

the expected discounted return from the initial-state distribution d_0, and improve it by *gradient ascent*,

$$\theta \leftarrow \theta + \eta \, \nabla_\theta J(\theta),$$

with a step size $\eta > 0$. In words: nudge the parameters in the direction that increases the expected return the fastest. In an episodic task the sum stops at the final time T; formally, the episode enters an absorbing terminal state where all rewards are zero, so the infinite sum and the finite one agree.

The whole difficulty is computing $\nabla_\theta J(\theta)$. The return depends on θ through every action the policy takes and, through those actions, through every state the agent visits. We do not know the transition probabilities, so we cannot differentiate them. Chapter 2 solved the one-state version of this problem: for a bandit with softmax preferences, the gradient bandit update $h \leftarrow h + \eta\,(r - \bar{r})(\mathbb{1}\{a\} - \pi)$ is a stochastic estimate of the gradient of the expected reward, with the average reward $\bar{r}$ as a baseline (Section 2.6). Everything in this chapter generalizes that update to problems with states and delayed consequences.

KEY IDEA

A policy gradient method represents the policy itself, $\pi_\theta(a \mid s)$, and improves it by stochastic gradient ascent on the expected return $J(\theta)$. It needs no maximization over actions, handles continuous actions and stochastic policies naturally, and changes its behavior smoothly.

11.2 Policy parameterizations

Before we can differentiate J we need policies with parameters. Two families cover almost all practice: softmax policies for discrete actions and Gaussian policies for continuous ones.

11.2.1 Softmax policies for discrete actions

Give every action in every state a real-valued *preference*, or *logit*, $h_\theta(s, a)$, and turn the preferences into probabilities with the softmax function:

$$\pi_\theta(a \mid s) = \frac{\exp\big(h_\theta(s, a)\big)}{\sum_{b \in \mathcal{A}} \exp\big(h_\theta(s, b)\big)}.$$

Larger preferences mean larger probabilities, every action keeps a positive probability, and adding the same constant to all preferences in a state changes nothing. The preferences can be a table, $h_\theta(s, a) = \theta_{s,a}$ (one parameter per state and action); a linear function of features, $h_\theta(s, a) = \theta^\top x(s, a)$; or the outputs of a neural network that maps the state to one logit per action.

The quantity that every policy gradient method needs is the gradient of the log-probability of an action, called the *score* of the action:

$$\psi_\theta(s, a) = \nabla_\theta \log \pi_\theta(a \mid s).$$

For softmax policies it has a simple form. Since $\log \pi_\theta(a \mid s) = h_\theta(s,a) - \log \sum_b \exp(h_\theta(s,b))$, its derivative with respect to the preference of an action b is

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial h_\theta(s,b)} = \mathbb{1}\{a = b\} - \pi_\theta(b \mid s). \tag{11.2}$$

In words: increasing the score direction raises the preference of the chosen action and lowers all preferences in proportion to their current probabilities. For linear preferences the chain rule gives $\psi_\theta(s,a) = x(s,a) - \sum_b \pi_\theta(b \mid s)\, x(s,b)$: the features of the chosen action minus their average under the policy. For a neural network, (11.2) is the gradient that backpropagation sends into the logit layer. One property holds for every policy and will matter repeatedly: the score has mean zero,

$$\sum_a \pi_\theta(a \mid s)\, \psi_\theta(s,a) = \sum_a \nabla_\theta \pi_\theta(a \mid s) = \nabla_\theta \sum_a \pi_\theta(a \mid s) = \nabla_\theta 1 = 0.$$

With two actions, a softmax policy with preferences $(0, z)$ is the *logistic* policy $\pi(\text{second action}) = \sigma(z)$, where $\sigma(z) = 1/(1+e^{-z})$ is the sigmoid function. Its derivative is $\sigma'(z) = \sigma(z)(1-\sigma(z))$, and the scores with respect to z are $1-\sigma(z)$ for the second action and $-\sigma(z)$ for the first.

11.2.2 The Trail with logits

The Trail of Chapter 1 is the ideal test bed. In Start and Middle the hiker either goes on (go, reward 0) or leaves (exit, reward +1 in Start and +2 in Middle); in Goal the only action, finish, earns +4; every path ends in End. A policy is described by $p_0 = \pi(\text{go} \mid \text{Start})$ and $p_1 = \pi(\text{go} \mid \text{Middle})$. We now give each decision a logit,

$$p_0 = \sigma(z_0), \qquad p_1 = \sigma(z_1), \qquad \theta = (z_0, z_1) \in \mathbb{R}^2,$$

so that every pair of logits is a valid policy and the uniform policy is $z = (0,0)$. Chapter 1 computed the expected return by listing the three paths: exit at once (return 1), go and exit at Middle (return 2γ), and go all the way (return $4\gamma^2$). Hence

$$J(z) = (1-p_0)\cdot 1 + p_0(1-p_1)\cdot 2\gamma + p_0 p_1 \cdot 4\gamma^2. \tag{11.3}$$

With $\gamma = 0.9$ the uniform policy earns $J = 1.76$ and the best policy, which always goes, earns 3.24.

Example 11.1 The Trail's gradient by hand

Take the uniform policy, $z = (0,0)$, so $p_0 = p_1 = \frac{1}{2}$, and $\gamma = 0.9$. We differentiate (11.3) with the chain rule, $\partial J/\partial z_i = (\partial J/\partial p_i)\,\sigma'(z_i)$, where $\sigma'(z_i) = p_i(1-p_i) = 0.25$.

The decision at Start. Collecting the terms with p_0,

$$\frac{\partial J}{\partial p_0} = -1 + (1-p_1)\,2\gamma + p_1\, 4\gamma^2 = -1 + 0.9 + 1.62 = 1.52.$$

This is a difference of action values. Going on from Start is worth $Q(\text{Start}, \text{go}) = \gamma\, V(\text{Middle}) = 0.9 \times 2.8 = 2.52$ and exiting is worth $Q(\text{Start}, \text{exit}) = 1$; their difference is 1.52.

The decision at Middle. Only the last two terms contain p_1:

$$\frac{\partial J}{\partial p_1} = p_0\,(4\gamma^2 - 2\gamma) = \gamma\, p_0\,(4\gamma - 2) = 0.9 \times 0.5 \times 1.6 = 0.72.$$

Again an action-value difference appears, $Q(\text{Middle}, \text{go}) - Q(\text{Middle}, \text{exit}) = 4\gamma - 2 = 3.6 - 2 = 1.6$, now multiplied by $\gamma p_0 = 0.45$, the discounted probability of being in Middle at all.

Multiplying by $\sigma'(z_i) = 0.25$,

$$\nabla_z J = (0.25 \times 1.52,\ 0.25 \times 0.72) = (0.38,\ 0.18).$$

Each component is a product of three factors:

Decision	discounted chance of reaching it	sensitivity $p(1-p)$	$Q(\text{go}) - Q(\text{exit})$	gradient
Start	1	0.25	1.52	0.38
Middle	$\gamma p_0 = 0.45$	0.25	1.60	0.18

The gradient says: going on is better than exiting in both states, so raise both logits, and raise z_0 faster, because Start is visited in every episode while Middle is reached only half the time and one step later.

♦

The three factors of Example 11.1 are not an accident of the Trail. The policy gradient theorem of the next section says that, in every problem, the gradient is a sum over states of exactly such products: how often (with discounting) the policy visits the state, how sensitive its action probabilities are to the parameters, and how much better one action is than another.

Figure 11.1 shows the whole landscape. In probability coordinates (left) the objective is gently curved and the gradient points toward the corner $p_0 = p_1 = 1$ everywhere. In logit coordinates (right), the coordinates the algorithm actually moves in, large regions are almost flat. Where a policy is nearly deterministic, $p(1-p) \approx 0$ and the gradient nearly vanishes, *whether or not the policy is good*. A policy that almost always exits at Start (say $z_0 = -5$, so $p_0 \approx 0.007$) earns about 1 and has a gradient of about 0.01: it is stuck on a plateau. Chapter 12 shows how natural gradients remove this effect, and Chapter 14 studies how long plain gradient methods can stay on such plateaus.

11.2.3 Gaussian policies for continuous actions

When the action is a real number, such as a torque, the standard choice is a *Gaussian policy*: in state s the action is drawn from a normal distribution whose mean $\mu_\theta(s)$ is computed by a network (or a linear function) and whose standard deviation is σ:

$$\pi_\theta(a \mid s) = \frac{1}{\sigma\sqrt{2\pi}} \exp\!\Big(-\frac{(a - \mu_\theta(s))^2}{2\sigma^2}\Big), \qquad \log \pi_\theta(a \mid s) = -\frac{(a - \mu_\theta(s))^2}{2\sigma^2} - \log\sigma - \tfrac{1}{2}\log(2\pi).$$

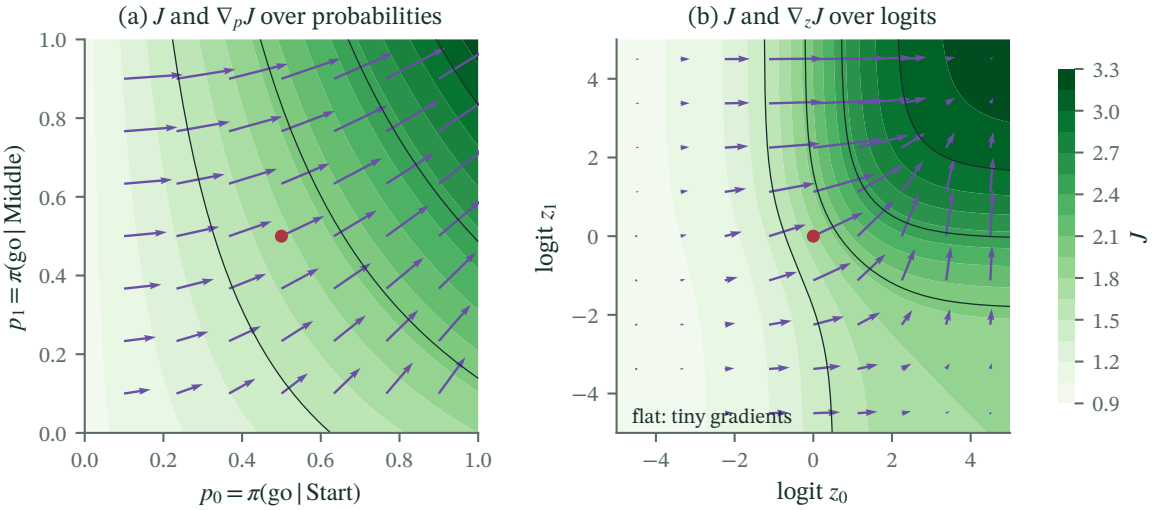


Figure 11.1. The Trail's objective J ($\gamma = 0.9$) and its gradient field. (a) Over the probabilities (p_0, p_1), the gradient points toward the optimal corner $(1, 1)$ from everywhere. (b) Over the logits (z_0, z_1), where the algorithm moves, the arrows shrink to almost nothing wherever the policy is close to deterministic, including the bad region on the left where the hiker almost always exits at Start. The dot marks the uniform policy, where $\nabla_z J = (0.38, 0.18)$.

Differentiating the log-density gives the two scores

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \mu} = \frac{a - \mu_\theta(s)}{\sigma^2}, \qquad \frac{\partial \log \pi_\theta(a \mid s)}{\partial \log \sigma} = \frac{(a - \mu_\theta(s))^2}{\sigma^2} - 1,$$

and the chain rule carries the first one into the network: $\nabla_\theta \log \pi_\theta(a \mid s) = \frac{a - \mu_\theta(s)}{\sigma^2} \nabla_\theta \mu_\theta(s)$. The scores are easy to read. If the sampled action lay above the mean and turned out well, the mean moves up toward it. If the action was far from the mean (more than one standard deviation) and turned out well, the standard deviation grows, since exploring widely paid off; if it was close to the mean, the standard deviation shrinks.

The standard deviation is usually parameterized by its logarithm, so that it stays positive without constraints. It can be a separate parameter vector that does not depend on the state (the common choice in PPO, Chapter 12) or a second output of the network. For actions with several components one uses a Gaussian with a diagonal covariance, and the log-density is the sum of the components' log-densities. Section 11.8 discusses what to do when actions must lie in a bounded interval.

CHECKPOINT

1. A softmax policy has logits $(1, 2, 3)$ in some state. If action 3 is sampled, what is its score with respect to the three logits?
2. In Example 11.1, why does Middle's gradient component contain the factor p_0?
3. A Gaussian policy has $\mu = 0.5$ and $\sigma = 0.2$, and samples $a = 0.9$. What are the two scores?

Answers.

1. The probabilities are $\mathrm{softmax}(1,2,3) = (0.090, 0.245, 0.665)$, so the score is $\mathbb{1}\{a = b\} - \pi(b) = (-0.090, -0.245, 0.335)$.
2. The decision at Middle only matters in episodes that reach Middle, which happens with probability p_0; the effect is also delayed by one step, which contributes the factor γ.
3. With respect to the mean, $(0.9 - 0.5)/0.04 = 10$; with respect to $\log\sigma$, $(0.4/0.2)^2 - 1 = 3$.

11.3 The policy gradient theorem

We now compute $\nabla_\theta J$ for any problem, without knowing the transition probabilities. The tool is a one-line identity from probability theory; the result is the central theorem of this chapter.

11.3.1 The score-function trick

Suppose a random quantity x is drawn from a distribution $p_\theta(x)$ that depends on parameters θ, and we want the gradient of the expectation of some function $f(x)$. If x takes finitely many values, the derivative only touches the probabilities, and $\nabla_\theta p_\theta(x) = p_\theta(x)\,\nabla_\theta \log p_\theta(x)$ turns the result back into an expectation:

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta}\big[f(x)\big] = \sum_x f(x)\,\nabla_\theta p_\theta(x) = \sum_x p_\theta(x)\,f(x)\,\nabla_\theta \log p_\theta(x) = \mathbb{E}_{x\sim p_\theta}\big[f(x)\,\nabla_\theta \log p_\theta(x)\big]. \tag{11.4}$$

For densities the sums become integrals, and the identity holds whenever we may differentiate under the integral sign, which is the case for Gaussian policies and bounded f. In words: *the gradient of an expectation is the expectation of the function times the score.* This is the *score-function* or *likelihood-ratio* trick. Its great virtue is that f is never differentiated. It can be a black box, observed only through samples, such as the return produced by an unknown environment.

Apply it to a whole episode. A trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ has probability

$$p_\theta(\tau) = d_0(s_0)\prod_{t\ge 0} \pi_\theta(a_t \mid s_t)\,P(s_{t+1} \mid s_t, a_t),$$

so its log-probability is a sum, and only the policy terms depend on θ:

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t\ge 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t).$$

The transition probabilities have disappeared: they do not depend on θ. Taking $f(\tau) = G_0$, the return of the episode, (11.4) gives

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\, G_0 \sum_{t\ge 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\Big].$$

This is already a usable formula: run an episode, add up the scores of the actions taken, multiply by the return. It can be improved by one observation. The action a_t cannot influence the rewards $r_0, \dots, r_{t-1}$ that were received before it was taken, so those rewards contribute only noise to the coefficient of its score. Removing them, and writing what remains in terms of the return from time t, gives the form that every algorithm uses.

11.3.2 The theorem

Recall the *discounted state-visitation distribution* of Definition 3.8, $d^{\pi}(s) = (1-\gamma)\sum_{t\geq 0}\gamma^t \Pr_{\pi}(s_t = s)$, a probability distribution over states that weights early visits more than late ones.

Theorem 11.1 Policy gradient theorem

Let the MDP have finitely many states and actions, discount $\gamma \in [0,1)$ and initial-state distribution d_0, and let $\pi_\theta(a \mid s)$ be differentiable in $\theta \in \mathbb{R}^d$ for every s and a. Then $J(\theta) = \mathbb{E}_{s_0\sim d_0}[V^{\pi_\theta}(s_0)]$ is differentiable, and

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\sum_{t=0}^{\infty}\gamma^t\, Q^{\pi_\theta}(s_t,a_t)\,\nabla_\theta \log \pi_\theta(a_t \mid s_t)\Big]$$
$$= \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d^{\pi_\theta},\, a\sim\pi_\theta(\cdot|s)}\big[Q^{\pi_\theta}(s,a)\,\nabla_\theta \log \pi_\theta(a \mid s)\big].$$

Actions with $\pi_\theta(a \mid s) = 0$ are never sampled and contribute nothing. Episodic tasks are included: their terminal states are absorbing with zero reward.

In words: the gradient is the expected score of the actions the policy takes, each weighted by how good the action is, $Q^{\pi_\theta}(s_t,a_t)$, and discounted by how late it is taken. What is remarkable is what is missing. Changing θ changes which states the agent visits, and the visitation distribution d^{π_θ} depends on θ in a complicated way through the unknown dynamics. Yet no derivative of d^{π_θ} appears: to estimate the gradient, the agent only needs to sample states from its own behavior and know the scores of its own actions.

Check the theorem on the Trail at the uniform policy, using the second form multiplied out: the weights $\sum_t \gamma^t \Pr(s_t = s)$ are 1 for Start and $\gamma p_0 = 0.45$ for Middle (Goal has no parameter). In Start the score with respect to z_0 is $+0.5$ for go and -0.5 for exit, so

$$\textstyle\sum_a \pi(a \mid \text{Start})\, Q(\text{Start},a)\, \psi_{z_0} = 0.5\times 2.52\times 0.5 + 0.5\times 1\times(-0.5) = 0.38,$$

and in Middle, $0.45\times(0.5\times 3.6\times 0.5 + 0.5\times 2\times(-0.5)) = 0.45\times 0.4 = 0.18$. This is the gradient $(0.38, 0.18)$ of Example 11.1, now obtained without differentiating J.

> **PITFALL**
>
> The policy gradient theorem is an *on-policy* statement: the expectation is over states and actions generated by π_θ itself. Episodes collected by an earlier policy, or stored in a replay buffer, do not estimate $\nabla_\theta J(\theta)$ unless they are reweighted, for example with importance-sampling ratios. Chapter 12 shows how to reuse slightly stale data safely, and Chapter 13 develops off-policy actor-critic methods.

11.3.3 Proof via trajectories ●●

We give two proofs. The first follows the score-function argument above and shows exactly why rewards from before an action drop out. For this proof we assume in addition that $\pi_\theta(a \mid s) > 0$

for all s, a and θ (true for softmax and Gaussian policies) and that π_θ is continuously differentiable, so that the scores $\psi_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ are bounded on a compact neighborhood $\mathcal{N}$ of θ, say by Ψ. Rewards are bounded, $|r(s,a)| \le r_{\max}$, because the MDP is finite.

Proof. *Step 1: one reward at a time.* Fix t and let $h_t = (s_0, a_0, \dots, s_t, a_t)$ be the history up to the action at time t. Its probability is $\Pr_\theta(h_t) = d_0(s_0)\prod_{k=0}^{t}\pi_\theta(a_k \mid s_k)\prod_{k=0}^{t-1}P(s_{k+1} \mid s_k, a_k)$, and $\mathbb{E}_\theta[r_t] = \sum_{h_t}\Pr_\theta(h_t)\,r(s_t,a_t)$ is a finite sum. By the score-function trick, $\nabla_\theta \Pr_\theta(h_t) = \Pr_\theta(h_t)\sum_{k=0}^{t}\psi_k$, so

$$\nabla_\theta \mathbb{E}_\theta[r_t] = \mathbb{E}_\theta\Big[r_t \sum_{k=0}^{t}\psi_k\Big].$$

Only the scores of actions taken *up to time t* appear, because the probability of h_t does not involve later actions. This is the precise sense in which an action cannot affect earlier rewards.

Step 2: all rewards. $J(\theta) = \sum_{t\ge 0}\gamma^t\mathbb{E}_\theta[r_t]$. On $\mathcal{N}$ the derivative of the t-th term is bounded by $\gamma^t r_{\max}(t+1)\Psi$, and $\sum_t (t+1)\gamma^t < \infty$, so the series of derivatives converges uniformly on $\mathcal{N}$. Hence J is differentiable and $\nabla_\theta J = \sum_{t\ge 0}\gamma^t\mathbb{E}_\theta\big[r_t\sum_{k\le t}\psi_k\big]$.

Step 3: regroup by action. The double series $\sum_t\sum_{k\le t}\gamma^t\mathbb{E}[r_t\psi_k]$ converges absolutely by the same bound, so we may sum over t first for each fixed k:

$$\nabla_\theta J = \sum_{k\ge 0}\mathbb{E}_\theta\Big[\psi_k\sum_{t\ge k}\gamma^t r_t\Big] = \sum_{k\ge 0}\gamma^k\,\mathbb{E}_\theta[\psi_k\,G_k], \qquad G_k = \sum_{t\ge k}\gamma^{t-k}r_t,$$

where moving the sum over t inside the expectation is justified by dominated convergence ($|\sum_{t\ge k}\gamma^t r_t\psi_k| \le \Psi r_{\max}/(1-\gamma)$).

Step 4: condition on the history. ψ_k is a function of h_k, and by the Markov property $\mathbb{E}_\theta[G_k \mid h_k] = Q^{\pi_\theta}(s_k,a_k)$. By the tower rule, $\mathbb{E}_\theta[\psi_k G_k] = \mathbb{E}_\theta[\psi_k\,Q^{\pi_\theta}(s_k,a_k)]$, which proves the first form (dominated convergence moves the sum over k back inside the expectation). For the second form, write $\phi(s) = \sum_a \pi_\theta(a \mid s)\,Q^{\pi_\theta}(s,a)\,\nabla_\theta\log\pi_\theta(a \mid s)$; then $\mathbb{E}_\theta[\psi_k Q^{\pi_\theta}(s_k,a_k)] = \sum_s \Pr_\theta(s_k = s)\,\phi(s)$, and summing γ^k times this over k gives $\sum_s \frac{d^{\pi_\theta}(s)}{1-\gamma}\,\phi(s)$. ∎

11.3.4 Proof via the Bellman equation ●●

The second proof differentiates the Bellman equation instead of the trajectory distribution. It needs neither positive probabilities nor any limit argument, and it shows that the gradient of the value function satisfies a Bellman equation of its own.

Proof. *Step 1: differentiability.* Write $P_\theta(s,s') = \sum_a \pi_\theta(a \mid s)P(s' \mid s,a)$ and $r_\theta(s) = \sum_a \pi_\theta(a \mid s)\,r(s,a)$. By (3.9), $V^{\pi_\theta} = (I - \gamma P_\theta)^{-1} r_\theta$. The entries of P_θ and r_θ are differentiable in θ, $I - \gamma P_\theta$ is invertible for every θ, and the inverse of a matrix is a differentiable function of its entries wherever it exists; so V^{π_θ} is differentiable, and so is $Q^{\pi_\theta}(s,a) = r(s,a) + \gamma\sum_{s'}P(s' \mid s,a)V^{\pi_\theta}(s')$.

Step 2: differentiate the Bellman equation. Drop the superscripts and differentiate $V(s) = \sum_a \pi_\theta(a \mid s)\,Q(s,a)$ with the product rule. The rewards do not depend on θ, so $\nabla_\theta Q(s,a) =$

$\gamma \sum_{s'} P(s' \mid s, a)\, \nabla_\theta V(s')$, and

$$\nabla_\theta V(s) = \underbrace{\sum_a \nabla_\theta \pi_\theta(a \mid s)\, Q(s,a)}_{\phi(s)} + \gamma \sum_{s'} P_\theta(s, s')\, \nabla_\theta V(s').$$

Step 3: solve it. For each component of θ this is a Bellman equation, with $\phi(s)$ in the role of the reward. Its unique solution is $\nabla_\theta V = (I - \gamma P_\theta)^{-1}\phi = \sum_{t \geq 0} \gamma^t P_\theta^t\, \phi$, that is,

$$\nabla_\theta V(s) = \sum_{t \geq 0} \gamma^t \sum_{s'} \Pr_\theta(s_t = s' \mid s_0 = s)\, \phi(s').$$

Step 4: average over the start. $\nabla_\theta J = \sum_{s_0} d_0(s_0) \nabla_\theta V(s_0) = \sum_s \big(\sum_t \gamma^t \Pr_\theta(s_t = s)\big)\phi(s) = \frac{1}{1-\gamma} \sum_s d^{\pi_\theta}(s)\, \phi(s)$. Finally, for actions with $\pi_\theta(a \mid s) > 0$, $\nabla_\theta \pi_\theta(a \mid s) = \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s)$, and for an action with $\pi_\theta(a \mid s) = 0$ the gradient $\nabla_\theta \pi_\theta(a \mid s)$ vanishes, because a differentiable function has zero gradient at a minimum. So $\phi(s) = \sum_a \pi_\theta(a \mid s)\, Q(s,a)\, \nabla_\theta \log \pi_\theta(a \mid s)$, with zero-probability actions omitted. The first form follows from $\sum_t \gamma^t \mathbb{E}_\theta[\phi(s_t)]$ and the tower rule. ■

Step 3 deserves a sentence in words: *the gradient of the value is the value of the local gradient*. Each state s' contributes its local improvement direction $\phi(s')$, weighted by how often and how soon the policy reaches it. Exercise 11.10 carries out this computation for the Trail.

11.4 REINFORCE

The policy gradient theorem expresses $\nabla_\theta J$ as an expectation under the agent's own behavior, so we can estimate it by sampling. The quantity $Q^{\pi_\theta}(s_t, a_t)$ inside it is unknown, but the return G_t that actually followed (s_t, a_t) is an unbiased sample of it: $\mathbb{E}[G_t \mid s_t, a_t] = Q^{\pi_\theta}(s_t, a_t)$. Replacing one by the other gives the oldest policy gradient algorithm, REINFORCE (Williams 1992).

Algorithm 11.1 REINFORCE (Monte Carlo policy gradient)

Input: a differentiable policy π_θ, step size $\eta > 0$, discount γ.

1. Initialize θ.
2. Repeat for each episode:
   1. Generate an episode $s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}$ by following π_θ.
   2. For $t = T-1, \ldots, 0$: $G_t \leftarrow r_t + \gamma G_{t+1}$, with $G_T = 0$.
   3. $\theta \leftarrow \theta + \eta \sum_{t=0}^{T-1} \gamma^t\, G_t\, \nabla_\theta \log \pi_\theta(a_t \mid s_t)$.

Output: the policy parameters θ.

Step 2.2 computes the *rewards-to-go* backward in one pass. Step 2.3 moves the parameters along the estimate

$$\hat{g} = \sum_{t=0}^{T-1} \gamma^t \, G_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t),$$

whose expectation is $\nabla_\theta J(\theta)$ by Theorem 11.1. All scores are evaluated at the parameters that generated the episode, and the parameters change once per episode. In practice one averages $\hat{g}$ over a batch of episodes before each step, which reduces the noise in proportion to the batch size. The update has a plain reading: *make each action taken more likely, in proportion to the return that followed it*. Actions followed by large returns gain more probability than actions followed by small ones.

REINFORCE needs no model and no value function, it works with any differentiable policy, and its estimate is unbiased. Its weakness is variance. The return G_t is a sum of many random rewards, the score is random too, and their product can be large and of either sign.

Example 11.2 What REINFORCE sees on the Trail

At the uniform policy with $\gamma = 0.9$, an episode follows one of three paths, and each produces one estimate $\hat{g} = (\hat{g}_0, \hat{g}_1)$. Exiting at once has $G_0 = 1$ and score -0.5 for z_0, so $\hat{g} = (-0.5, 0)$. The path go, exit has $G_0 = 1.8$ with score $+0.5$ at Start, and $\gamma G_1 = 0.9 \times 2 = 1.8$ with score -0.5 at Middle, so $\hat{g} = (0.9, -0.9)$. Going all the way gives $\hat{g} = (3.24 \times 0.5,\ 0.9 \times 3.6 \times 0.5) = (1.62, 1.62)$.

Table 11.1. The three possible single-episode REINFORCE estimates at the uniform policy.

Path	probability	estimate $\hat{g}$
exit at Start	0.5	$(-0.5,\ 0)$
go, exit	0.25	$(0.9,\ -0.9)$
go, go, finish	0.25	$(1.62,\ 1.62)$

The average is $0.5\,(-0.5, 0) + 0.25\,(0.9, -0.9) + 0.25\,(1.62, 1.62) = (0.38, 0.18)$, exactly the gradient of Example 11.1. No single episode gives the right answer, and half of all episodes push z_0 the *wrong way*: exiting earned a positive return, so its probability is raised. The variances of the two components are 0.839 and 0.826, so the total variance (the sum of the component variances) is 1.665, and a single estimate is typically off by about 0.9 in each component, more than twice the size of the gradient itself.

♦

Despite the noise, REINFORCE learns. With one episode per update and $\eta = 0.1$, it takes the uniform Trail policy to $p_0, p_1 > 0.9$ in a median of 105 episodes (over 100 runs; 90% of them within 152). The red points in Figure 11.2 show the averages of batches of ten episodes: they scatter widely around the true gradient. The next section shows how to shrink that cloud.

HISTORICAL NOTE

Williams (1992) introduced REINFORCE for networks of stochastic units; the name abbreviates "REward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility", which describes the update with a baseline. The likelihood-ratio idea was developed independently for the simulation of stochastic systems (Glynn 1990). The policy gradient theorem in the form used here, valid with function approximation, is due to Sutton et al. (2000), with related results by Marbach and Tsitsiklis (2001) and by Baxter and Bartlett (2001).

11.5 Baselines

On CartPole every reward is $+1$, so every return is positive. REINFORCE therefore raises the probability of *every* action it takes, only by more for actions followed by long episodes. The useful information, which actions led to *longer than usual* episodes, is a small difference riding on a large common offset, and the offset contributes nothing but noise. The remedy is to subtract it.

Theorem 11.2 Baselines do not bias the policy gradient

Let $b(s)$ be any function of the state (it may also depend on the time t and on the history before a_t, but not on a_t or on anything that happens after it). Then

$$\mathbb{E}_{\pi_\theta}\big[\gamma^t\, b(s_t)\, \nabla_\theta \log \pi_\theta(a_t \mid s_t)\big] = 0 \quad \text{for every } t,$$

and therefore

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\sum_{t\ge 0} \gamma^t \big(G_t - b(s_t)\big) \nabla_\theta \log \pi_\theta(a_t \mid s_t)\Big].$$

Proof. Condition on everything up to and including s_t. Given that information, $b(s_t)$ is a fixed number and $a_t \sim \pi_\theta(\cdot \mid s_t)$, so

$$\mathbb{E}\big[b(s_t)\, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \,\big|\, s_t\big] = b(s_t) \sum_a \pi_\theta(a \mid s_t)\, \nabla_\theta \log \pi_\theta(a \mid s_t) = b(s_t)\, \nabla_\theta \sum_a \pi_\theta(a \mid s_t) = 0,$$

because the probabilities sum to one for every θ. By the tower rule the unconditional expectation is zero as well. The second statement follows by subtracting these zero terms from the estimate $\sum_t \gamma^t G_t \psi_t$, whose expectation is $\nabla_\theta J$. For continuous actions the sums become integrals; the argument is the same. ∎

A baseline changes the variance but never the mean, so we may choose it freely to make the variance small. The natural choice is the state value, $b(s) = V^{\pi_\theta}(s)$. Then the coefficient of the score becomes $G_t - V^{\pi_\theta}(s_t)$, a sample of the *advantage* $A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$, which answers the question that matters: was this action better or worse than what the policy usually does in this state? Actions that did better than usual become more likely; actions that did worse become less likely.

Example 11.3 A baseline on the Trail

Use the uniform policy's values as the baseline: $b(\text{Start}) = 1.76$ and $b(\text{Middle}) = 2.8$. Exiting at once now gives $(1 - 1.76) \times (-0.5) = 0.38$ for z_0: the return was *below* average, so lowering the probability of exiting is the right lesson. The path go, exit gives $(1.8 - 1.76) \times 0.5 = 0.02$ for z_0 and $0.9 \times (2 - 2.8) \times (-0.5) = 0.36$ for z_1. Going all the way gives $(3.24 - 1.76) \times 0.5 = 0.74$ and $0.9 \times (3.6 - 2.8) \times 0.5 = 0.36$. The three estimates are

$$(0.38,\ 0), \qquad (0.02,\ 0.36), \qquad (0.74,\ 0.36),$$

with the same probabilities 0.5, 0.25, 0.25. Their mean is still $(0.38, 0.18)$, but every estimate now has the right sign in every component, and the total variance falls from 1.665 to 0.097, a factor of about 17.

♦

Figure 11.2 shows the effect on batches of ten episodes. With the baseline (green) the estimates crowd around the true gradient; without it (red) they spread over a region several times larger than the gradient.

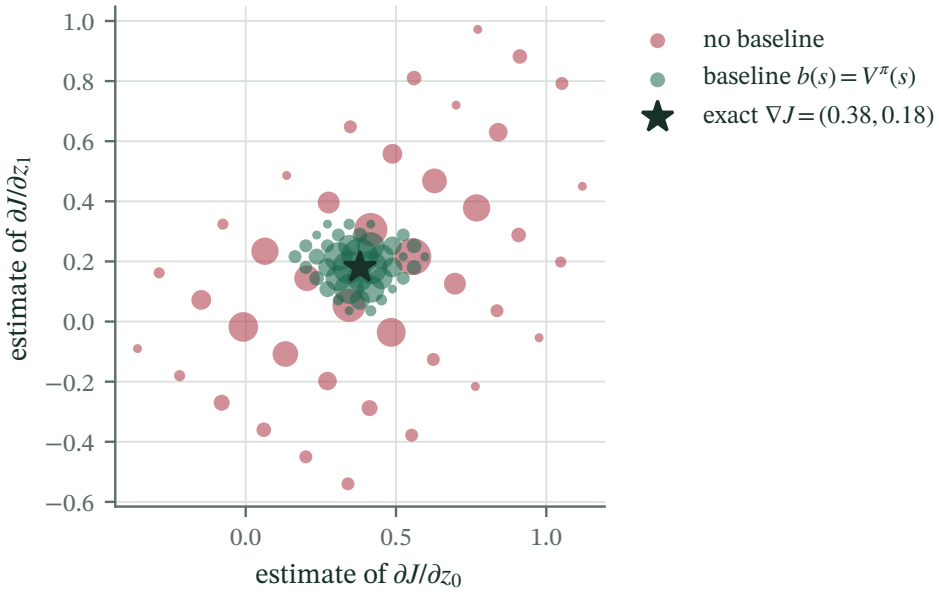


Figure 11.2. REINFORCE estimates of the Trail's gradient at the uniform policy. Each point is the average of a batch of ten episodes (300 batches per method); the area of a marker is proportional to how many batches gave that value, since only a few distinct values are possible. Both clouds are centered on the exact gradient $(0.38, 0.18)$ (star), but the value baseline shrinks the total variance of a batch estimate from 0.167 to 0.010 (the single-episode values 1.665 and 0.097 divided by ten).

In practice the agent does not know V^{π_θ}, so it learns it. A second function approximator, $V_w(s)$ with parameters w, is trained by regression on the observed returns, for instance with the gradient Monte Carlo update $w \leftarrow w + \eta_w(G_t - V_w(s_t))\nabla_w V_w(s_t)$ of Chapter 9. Because any baseline leaves the gradient unbiased, V_w does not need to be accurate: a poor baseline simply removes less variance. (A very poor one can add variance: with $b \equiv 10$ on the Trail the total variance grows to 22.3, Exercise 11.4.)

PITFALL

The baseline must not depend on the action whose score it multiplies, and it should not be computed from the very return it is subtracted from. If $b(s_t)$ is fitted to a batch that includes G_t itself, it is slightly correlated with a_t and the estimate acquires a small bias. Standard practice computes the policy step with the critic's values from before its update, and fits the critic afterward.

11.5.1 The optimal baseline ●●

Is the state value the best baseline? Not quite. Consider the contribution of one state s with a constant baseline b: the estimate $(G - b)\,\psi$, where $a \sim \pi_\theta(\cdot \mid s)$, $\psi = \nabla_\theta \log \pi_\theta(a \mid s)$ and G is the return that follows.

Proposition 11.3 The variance-minimizing constant baseline

Among constant baselines b, the total variance $\mathbb{E}\big[\|(G-b)\psi\|^2\big] - \|\mathbb{E}[G\psi]\|^2$ of the estimate $(G-b)\psi$ is minimized by

$$b^* = \frac{\mathbb{E}\big[\|\psi\|^2\, G\big]}{\mathbb{E}\big[\|\psi\|^2\big]}.$$

Proof. The mean $\mathbb{E}[(G - b)\psi] = \mathbb{E}[G\psi]$ does not depend on b (Theorem 11.2), so it suffices to minimize the second moment $\mathbb{E}[\|\psi\|^2 (G - b)^2]$, a quadratic function of b. Setting its derivative $-2\,\mathbb{E}[\|\psi\|^2 (G - b)]$ to zero gives b^*, and the second derivative $2\,\mathbb{E}\|\psi\|^2$ is positive. ■

The optimal baseline is a weighted average of returns, with weights proportional to the squared score. If $\|\psi\|^2$ is the same for every action, $b^* = \mathbb{E}[G] = V^{\pi_\theta}(s)$. In general actions with large scores, typically the unlikely ones, get more weight. On the Trail with $p_0 = 0.8$ and $p_1 = 0.5$, $V(\text{Start}) = 2.216$ but $b^* = 1.304$ for the Start component, and the variance of that component is 0.289 without a baseline, 0.150 with the value baseline and 0.017 with b^* (Exercise 11.8). Optimal baselines can be derived per parameter component and for the multi-step estimators (Weaver and Tao 2001; Greensmith et al. 2004; Peters and Schaal 2008b). Practice nonetheless uses learned state values: they are easy to learn, close enough to optimal in most problems, and they turn the estimate into an advantage, which the methods of the next sections need.

11.6 Actor-critic methods

A baseline removes the common offset of the returns, but not their own randomness. In a long episode, G_t sums hundreds of random rewards, and its variance grows with the horizon. And REINFORCE must wait until the end of an episode before it can learn anything, which is impossible in a continuing task. Both problems have the same solution as in Part II: *bootstrap*. Replace the return by one reward plus the estimated value of the next state.

With a learned state-value function V_w, the coefficient $G_t - V_w(s_t)$ becomes the temporal-difference error of (6.1),

$$\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t).$$

If the critic were exact, $V_w = V^{\pi_\theta}$, the TD error would be an unbiased sample of the advantage, because

$$\mathbb{E}\big[\delta_t \mid s_t, a_t\big] = r(s_t, a_t) + \gamma \sum_{s'} P(s' \mid s_t, a_t)\, V^{\pi_\theta}(s') - V^{\pi_\theta}(s_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t).$$

With an approximate critic it is biased, but its variance is far smaller: it contains one random reward and one random next state instead of a whole future. The policy is now called the *actor* and the value function the *critic*: the actor acts, and the critic judges each action by whether it led to a better or worse state than expected. The names and the architecture go back to Barto et al. (1983), whose actor-critic system learned to balance a pole.

Algorithm 11.2 One-step actor-critic (episodic)

Input: a differentiable policy π_θ, a differentiable critic V_w, step sizes $\eta_\theta, \eta_w > 0$, discount γ.

1. Initialize θ and w.
2. For each episode:
   1. Observe the initial state s; set $I \leftarrow 1$.
   2. Repeat until s is terminal:
      1. Sample $a \sim \pi_\theta(\cdot \mid s)$; take it; observe r and s'.
      2. $\delta \leftarrow r + \gamma V_w(s') - V_w(s)$, with $V_w(s') = 0$ if s' is terminal.
      3. $w \leftarrow w + \eta_w\, \delta\, \nabla_w V_w(s)$.
      4. $\theta \leftarrow \theta + \eta_\theta\, I\, \delta\, \nabla_\theta \log \pi_\theta(a \mid s)$.
      5. $I \leftarrow \gamma I$; $s \leftarrow s'$.

Output: the actor π_θ and the critic V_w.

The critic's update (step 2.2.3) is semi-gradient TD(0) from Chapter 9. The actor's update (step 2.2.4) is the policy gradient step with the TD error as the advantage estimate; the factor $I = \gamma^t$ is the γ^t of the policy gradient theorem. If the episode is cut off by a time limit rather than ended, $V_w(s')$ is *not* set to zero (Section 6.8).

Actor-critic methods learn online, after every step, and they work in continuing tasks. They are not always faster. On the Trail, with a tabular critic that starts at zero and step sizes $\eta_\theta = \eta_w = 0.1$, one-step actor-critic needs a median of 212 episodes to reach $p_0, p_1 > 0.9$, twice as many as REINFORCE. Early on, the zero critic says that going on from Start leads nowhere ($\delta = 0 + \gamma \cdot 0 - 0 = 0$) while exiting earns $\delta = 1$, so the actor first learns to exit; it recovers once the critic has learned the values (Exercise 11.11). With a faster critic ($\eta_w = 0.3$) the median drops to 143. On a three-step problem with cheap episodes, Monte Carlo returns are hard to beat; the advantages of bootstrapping appear with long horizons, and the bias–variance balance between the two can be tuned continuously, as Section 11.7 shows.

Between the one-step TD error and the full return lie the n-step returns of Chapter 7. Using $G_{t:t+n} - V_w(s_t)$ as the advantage estimate, where $G_{t:t+n}$ is the n-step return of (7.1) bootstrapped from $V_w(s_{t+n})$, gives an actor-critic method that waits n steps before updating.

11.6.1 Parallel environments: A3C and A2C

Deep actor-critic methods face one more problem. Consecutive transitions from one environment are strongly correlated, so a minibatch made of them is a poor sample of the state distribution, and the network overfits to whatever part of the state space it is currently in. DQN fixed this with a replay buffer (Chapter 10), but replayed transitions come from older policies, and the policy gradient theorem needs on-policy data. Mnih et al. (2016) proposed a different fix: run many copies of the environment in parallel, so that each batch mixes experience from many independent episodes. Their *asynchronous advantage actor-critic* (A3C) ran actor-learner threads that each computed gradients on their own environment and applied them asynchronously to shared parameters. The *synchronous* version, usually called A2C, steps all environments in lockstep and makes one update from the combined batch; it is simpler and uses hardware such as GPUs better.

Algorithm 11.3 A2C: synchronous advantage actor-critic

Input: N environments, rollout length n, discount γ, step sizes η_θ, η_w, entropy coefficient $c_H \geq 0$.

1. Initialize the actor π_θ and the critic V_w; reset all environments.
2. Repeat:
   1. For $t = 0, \dots, n-1$, in every environment i: sample $a_t^i \sim \pi_\theta(\cdot \mid s_t^i)$, take it, store $(s_t^i, a_t^i, r_t^i, s_{t+1}^i)$ and whether the episode terminated or was truncated; reset environments whose episodes ended.
   2. Compute an advantage estimate $\hat{A}_t^i$ for every stored step: the n-step return minus $V_w(s_t^i)$, bootstrapping from V_w at the end of the rollout and at truncations and from 0 at terminations (or use (11.5)).
   3. Set the critic targets $\hat{R}_t^i = \hat{A}_t^i + V_w(s_t^i)$.
   4. $\theta \leftarrow \theta + \eta_\theta \frac{1}{Nn} \sum_{i,t} \Big[\hat{A}_t^i \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) + c_H \nabla_\theta \mathcal{H}\big(\pi_\theta(\cdot \mid s_t^i)\big) \Big]$.
   5. $w \leftarrow w - \eta_w \frac{1}{Nn} \sum_{i,t} \big(V_w(s_t^i) - \hat{R}_t^i\big) \nabla_w V_w(s_t^i)$.

Output: the actor π_θ and the critic V_w.

The batch in step 2.4 has Nn transitions, for example $8 \times 32 = 256$. Each environment keeps running across updates, so an episode can span several rollouts; the bootstrapping in step 2.2 is what lets a rollout end in the middle of an episode. The entropy term in step 2.4, discussed in Section 11.8, keeps the policy from becoming deterministic too early. Like most implementations, step 2.4 omits the factor γ^t; Section 11.9 explains what this changes. In deep-learning code the two steps are usually written as one loss to be minimized,

$$\mathcal{L}(\theta, w) = -\frac{1}{Nn} \sum_{i,t} \Big[\hat{A}_t^i \log \pi_\theta(a_t^i \mid s_t^i) + c_H \, \mathcal{H}\big(\pi_\theta(\cdot \mid s_t^i)\big) \Big] + \frac{c_V}{2Nn} \sum_{i,t} \big(V_w(s_t^i) - \hat{R}_t^i\big)^2,$$

where the coefficient $c_V > 0$ weighs the critic's loss and the advantages and targets are treated as constants.

Figure 11.3 compares the three methods of this chapter on CartPole, all with small neural networks from the book's NumPy library and, as in most implementations, without the factor γ^t. A2C, updating every 256 steps from bootstrapped advantages, learns fastest at first: it averages about 330 steps per episode after 35,000 environment steps, when plain REINFORCE averages about 170 and REINFORCE with a baseline about 200. But its curve then flattens and wobbles, a typical consequence of a biased critic and of noisy updates on an on-policy method with no limit on how far each update moves the policy. REINFORCE with a learned baseline is slower at first, steadier later, and ends highest, with an average of about 470 after 150,000 steps against 414 for plain REINFORCE and 385 for A2C. No method dominates; the right choice depends on the horizon, the cost of samples and the tuning effort, and the next section adds a dial between them.

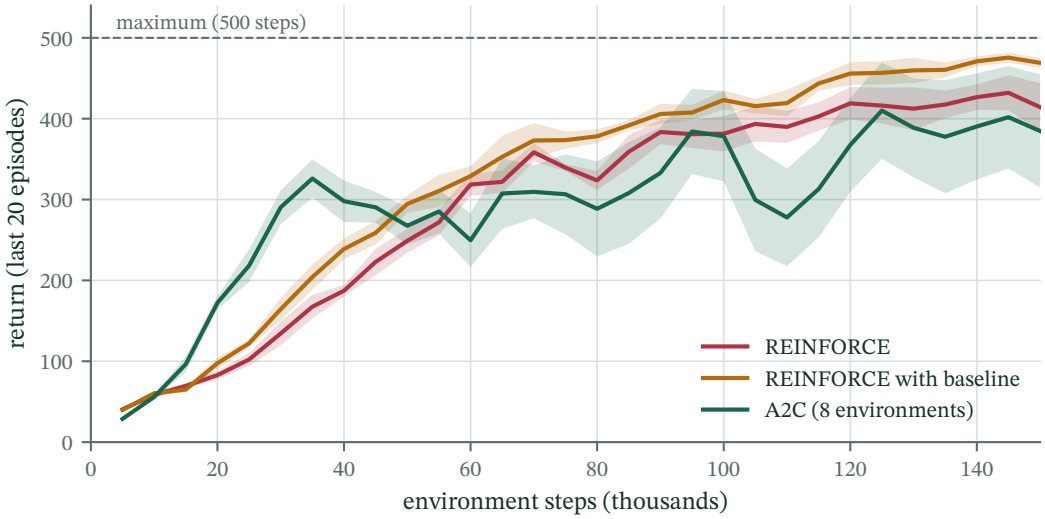


Figure 11.3. Learning curves on CartPole (the maximum return is 500). Each curve is the mean over five seeds of the average return of the last 20 episodes; the bands show ±1 standard error. REINFORCE uses batches of ten episodes; A2C uses eight parallel environments and rollouts of 32 steps. A2C learns fastest early on, while REINFORCE with a learned baseline is the steadiest and ends highest.

LAB

Lab 11.1: Policy gradients from the Trail to CartPole (`labs/ch11_pg.py`, about 45 minutes). Run `python3 labs/ch11_pg.py`. You will check the Trail's gradient by hand and by finite differences, enumerate the REINFORCE estimator and measure what a baseline does to its variance, train REINFORCE and one-step actor-critic on the Trail, compute GAE by hand, including a time-limit truncation, and train REINFORCE and A2C on CartPole. The "Going further" ideas include an entropy-bonus sweep.

TRY IT INTERACTIVELY

Module 6 of the interactive course (`Apps_claude/RL Course/index.html`) walks through a single REINFORCE update on the Trail, the γ^t factor, valid and invalid baselines, and GAE computed backward step by step.

11.7 Generalized advantage estimation

REINFORCE with a baseline uses the full return and has no bias but high variance; one-step actor-critic uses one reward and the critic, and has low variance but inherits the critic's errors. *Generalized advantage estimation* (GAE) puts a dial between the two (Schulman et al. 2016). It is the advantage estimator used by almost every modern on-policy method, including PPO.

Start from the n-step advantage estimates, which look n rewards ahead and then bootstrap from the critic V:

$$\hat{A}_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) - V(s_t).$$

Written with the TD errors $\delta_k = r_k + \gamma V(s_{k+1}) - V(s_k)$, the intermediate values cancel in pairs, and the estimate becomes a discounted sum of TD errors:

$$\hat{A}_t^{(n)} = \sum_{l=0}^{n-1} \gamma^l \, \delta_{t+l}.$$

(For $n = 2$: $\delta_t + \gamma\delta_{t+1} = r_t + \gamma V(s_{t+1}) - V(s_t) + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - \gamma V(s_{t+1})$, and the two $\gamma V(s_{t+1})$ terms cancel.) With $n = 1$ we get the one-step TD error of actor-critic; with n running to the end of the episode we get $G_t - V(s_t)$, REINFORCE with a baseline. Instead of choosing one n, GAE averages all of them with geometrically decreasing weights $(1-\lambda)\lambda^{n-1}$, exactly as the λ-return of (7.9) averages n-step returns. The result is remarkably simple:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \hat{A}_t^{(n)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}. \tag{11.5}$$

Proof. Substitute $\hat{A}_t^{(n)} = \sum_{l<n} \gamma^l \delta_{t+l}$ and exchange the order of summation: δ_{t+l} appears in every $\hat{A}_t^{(n)}$ with $n > l$, so its total weight is $\gamma^l(1-\lambda)\sum_{n>l}\lambda^{n-1} = \gamma^l\lambda^l$. In an episode that ends at time T, all δ_k with $k \geq T$ are zero (the terminal state has value zero and earns nothing), so the sums are finite. ∎

In words: the advantage of the action at time t is the TD error it caused, plus the later TD errors, faded by a factor $\gamma\lambda$ per step. Surprises that happen soon after the action are credited to it; surprises far in the future mostly are not. The parameter $\lambda \in [0,1]$ sets how far credit travels: $\lambda = 0$ gives $\hat{A}_t = \delta_t$, and $\lambda = 1$ gives $\hat{A}_t = \sum_l \gamma^l \delta_{t+l} = G_t - V(s_t)$. A rough horizon is $1/(1-\gamma\lambda)$ steps; for the common choice $\gamma = 0.99$, $\lambda = 0.95$, it is about 17. Equivalently, $\hat{A}_t^{\mathrm{GAE}} = G_t^\lambda - V(s_t)$, where G_t^λ is the λ-return built from n-step returns that bootstrap from V (Exercise 11.14); the critic's regression target $\hat{A}_t + V(s_t)$ is that λ-return.

Bias and variance. If the critic were exact, every $\hat{A}_t^{(n)}$ would have conditional mean $A^\pi(s_t, a_t)$, and so would GAE, for every λ. With an imperfect critic, λ trades the two kinds of error. Large λ uses many real rewards and is noisy. Small λ leans on the critic and inherits its errors. At $\lambda = 1$ something special happens: $\mathbb{E}[\hat{A}_t \mid s_t, a_t] = Q^\pi(s_t, a_t) - V(s_t)$, so the critic's error is only a state-dependent shift, a baseline, and by Theorem 11.2 it does not bias the policy gradient at all.

Figure 11.4 measures the trade-off exactly. The problem is the 4×3 grid world of Chapter 3 with $\gamma = 0.95$, a mediocre softmax policy (average episode length about 30 steps) and a critic that is

a policy behind: it holds the values of the uniform random policy. For each λ, the squared bias of the gradient estimate $\sum_t \gamma^t \hat{A}_t \nabla \log \pi$ is computed exactly with linear algebra, and its variance from 4,000 sampled episodes. Relative to $\|\nabla J\|^2$, the squared bias falls from 0.28 at $\lambda = 0$ to zero at $\lambda = 1$, while the variance of a one-episode estimate rises from 1.4 to 38, most of the rise happening above $\lambda = 0.9$. What matters for learning is the total error of the estimate actually used. With batches of 10 episodes the mean squared error is smallest near $\lambda = 0.5$; with batches of 100 episodes, near $\lambda = 0.85$. The more data per update, the less bias one can afford, consistent with the common choice $\lambda = 0.95$ in large-batch methods such as PPO.

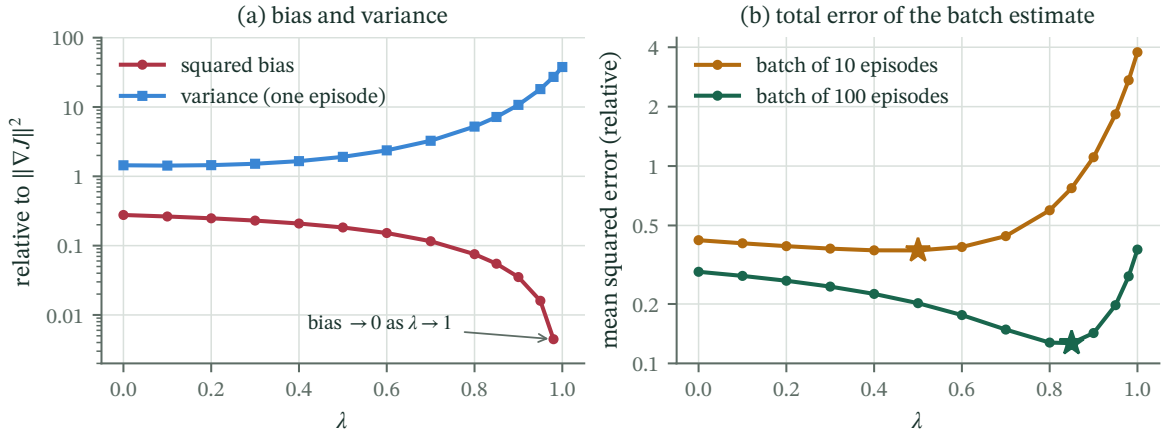


Figure 11.4. Bias and variance of GAE policy-gradient estimates on the 4×3 grid world ($\gamma = 0.95$), with a critic that still describes an earlier policy. (a) Squared bias (computed exactly) and variance of a single-episode estimate (from 4,000 episodes), both relative to $\|\nabla J\|^2$: the bias vanishes as $\lambda \to 1$ while the variance explodes. (b) The mean squared error of a batch average is the squared bias plus the variance divided by the batch size; stars mark the best λ, which moves toward 1 as batches grow.

Computing GAE. In code, (11.5) is computed backward over a rollout with the recursion

$$\hat{A}_t = \delta_t + \gamma\lambda\,(1 - e_t)\,\hat{A}_{t+1},$$

where $e_t = 1$ if the data segment ends after step t. The segment can end in three ways, and they must be handled differently in δ_t:

Table 11.2. How the end of a step enters the GAE computation.

How step t ends	value used for s_{t+1} in δ_t	does the sum continue past t?
the episode continues	$V(s_{t+1})$	yes
termination (terminal state)	0	no
truncation (time limit)	$V(s_{t+1})$ of the true final state	no
end of the rollout, mid-episode	$V(s_{t+1})$	no

Two details deserve emphasis. At a truncation the episode did not end in the MDP; it was cut off,

and the state reached still has a value, which must come from the *final* observation, not from the first observation of the next episode that a vectorized environment returns after its automatic reset. And at every boundary the sum must stop, or the next episode's TD errors leak into this one.

Example 11.4 GAE on the Trail

Use the uniform policy's values as the critic: $V(\text{Start}) = 1.76$, $V(\text{Middle}) = 2.8$, $V(\text{Goal}) = 4$, and $\gamma = 0.9$. In the episode go, go, finish the TD errors are

$$\delta_0 = 0 + 0.9 \times 2.8 - 1.76 = 0.76, \qquad \delta_1 = 0 + 0.9 \times 4 - 2.8 = 0.8, \qquad \delta_2 = 4 + 0 - 4 = 0.$$

With $\lambda = 0.5$, so $\gamma\lambda = 0.45$, the backward recursion gives $\hat{A}_2 = 0$, $\hat{A}_1 = 0.8$ and $\hat{A}_0 = 0.76 + 0.45 \times 0.8 = 1.12$. With $\lambda = 0$ the advantages are the TD errors $(0.76, 0.8, 0)$; with $\lambda = 1$, $\hat{A}_0 = 0.76 + 0.9 \times 0.8 = 1.48 = G_0 - V(\text{Start}) = 3.24 - 1.76$.

Now suppose a time limit of two steps cuts the episode when the hiker arrives at Goal. The correct computation treats Goal as a state with value 4: $\delta_1 = 0.9 \times 4 - 2.8 = 0.8$, and $\hat{A} = (1.12, 0.8)$ as before. The classic bug treats the cut as termination and bootstraps from zero: $\delta_1 = 0 - 2.8 = -2.8$ and $\hat{A} = (0.76 + 0.45 \times (-2.8), -2.8) = (-0.5, -2.8)$. The best action in Middle now looks terrible, and so does going on from Start.

♦

PITFALL

Truncation is not termination (Section 6.8). An episode cut off by a time limit must bootstrap from the value of the state where it was cut, and a vectorized environment that resets automatically must hand that final state to the advantage computation. Forgetting either is one of the most common bugs in policy gradient code, and it is silent: the agent still learns, only worse.

11.8 Practical policy gradients

The algorithms above are complete, but a handful of practical choices decide whether they work on a real problem. This section collects them; Chapter 12 adds the details specific to PPO, and Chapter 23 covers debugging and evaluation in general.

Entropy bonus. A policy gradient method can make its policy nearly deterministic long before it has found a good one, and then it is stuck on a plateau like those of Figure 11.1. Adding an entropy bonus to the objective, $J(\theta) + c_H \, \mathbb{E}_s\big[\mathcal{H}(\pi_\theta(\cdot \mid s))\big]$ with $\mathcal{H}(\pi(\cdot \mid s)) = -\sum_a \pi(a \mid s) \log \pi(a \mid s)$, rewards the policy for keeping its options open (Williams and Peng 1991; Mnih et al. 2016). For a softmax policy the entropy's gradient with respect to the logits is $\partial\mathcal{H}/\partial h(s,b) = -\pi(b \mid s)\big(\log \pi(b \mid s) + \mathcal{H}\big)$; for two actions with logit z it reduces to $-p(1-p)\,z$, a pull back toward $z = 0$ that is strongest for moderately confident policies (Exercise 11.7). The bonus changes the objective: with $c_H > 0$ the optimal policy is stochastic. Section 13.5 turns this into a principled framework, and Ahmed et al. (2019) study how entropy smooths the optimization landscape. Because the policy-gradient term scales with the advantages, c_H only has meaning relative to the advantages' size; values around 0.01 are common with normalized advantages.

Advantage normalization. Most implementations standardize the advantages of each batch, subtracting their mean and dividing by their standard deviation. Subtracting a constant is a baseline; dividing rescales the step so that it no longer depends on the scale of the rewards. Because the mean and standard deviation come from the same batch, the estimate is no longer exactly unbiased, but the effect is small for large batches, and the gain in robustness to reward scale is large. Its effect has been measured in large studies of on-policy methods (Andrychowicz et al. 2021).

Step sizes and batch sizes. The estimate's noise falls like $1/\sqrt{B}$ with the batch size B, and the step size must be small enough for the noise that remains. A step that is too large for its noise can push the policy into a near-deterministic region from which it never returns; Chapter 12 opens with exactly this failure. Adam with learning rates between 3×10^{-4} and 10^{-3} is a robust default for the small networks of this book's labs, together with clipping the global gradient norm (to 0.5 in our A2C runs).

Continuous actions. A Gaussian policy's standard deviation is best kept as a separate, state-independent parameter vector $\log \sigma$, initialized so that σ is a sizable fraction of the action range, and learned along with the mean. When the action must lie in an interval $[-a_{\max}, a_{\max}]$, there are two common options. The first samples from the Gaussian and lets the environment clip the action; the log-probabilities are those of the unclipped sample, which is consistent because the clipping is then part of the environment, but probability mass beyond the bounds is wasted. The second *squashes* the sample, $a = a_{\max} \tanh(u)$ with $u \sim \mathcal{N}(\mu_\theta(s), \sigma^2)$, and corrects the density by the change of variables, $\log \pi_\theta(a \mid s) = \log \mathcal{N}(u; \mu_\theta(s), \sigma^2) - \sum_j \log\bigl(a_{\max}(1 - \tanh^2 u_j)\bigr)$. Soft actor-critic uses the second (Algorithm 13.3); our Pendulum runs in Chapter 12 use the first.

Inputs and networks. Normalize observations with running means and standard deviations when their scales differ. Small networks (two hidden layers of 64 units) are enough for classic control. Separate actor and critic networks are the safe default; a shared body saves computation but couples the two losses, whose relative weight then becomes another hyperparameter.

Sanity checks. Test new code on a problem with a known gradient, such as the Trail: the estimator's average over many episodes must match the exact gradient and finite differences. During training, watch the policy's entropy (a collapse to zero is a warning), the critic's prediction error and the episode returns (Section 23.3).

CHECKPOINT

1. Which value of λ makes GAE equal to the one-step TD error, and which makes it equal to the Monte Carlo return minus the baseline?
2. A rollout ends in the middle of an episode. Which value does the last TD error use, and why must the sum stop there?
3. Why does an entropy bonus help a policy that has become nearly deterministic too early?

Answers.

1. $\lambda = 0$ gives δ_t; $\lambda = 1$ gives $G_t - V(s_t)$ (with bootstrapping from V at the end of a cut-off segment).
2. It uses $\gamma V(s_{t+1})$ of the state where the rollout stopped, because the episode goes on and that state has a value; the sum stops because the TD errors that follow have not been observed yet.

3. Near-deterministic softmax policies have nearly vanishing policy gradients; the entropy term adds a gradient that pushes the logits back toward a more random policy, where learning can resume.

11.9 Subtleties: discounting, average reward and gradient estimators ●●

11.9.1 The missing discount factor

Look again at the policy gradient theorem: the action taken at time t is weighted by γ^t. Almost every implementation drops this factor. REINFORCE, A2C and PPO as usually coded compute

$$\hat{g}_{\text{common}} = \sum_t \hat{A}_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t),$$

with advantages $\hat{A}_t$ that are discounted from time t onward but no γ^t in front. The reason is practical: with $\gamma = 0.99$, the factor γ^t makes everything after a few hundred steps nearly irrelevant to the update, which is rarely what the designer wants. But the resulting update is not the gradient of J.

What does it estimate? With exact advantages its expectation is

$$\mathbb{E}\big[\hat{g}_{\text{common}}\big] = \sum_s \eta^\pi(s)\, \phi(s), \qquad \eta^\pi(s) = \mathbb{E}_\pi\Big[\sum_t \mathbb{1}\{s_t = s\}\Big],$$

where $\phi(s) = \sum_a \pi_\theta(a \mid s) A^\pi(s,a) \nabla_\theta \log \pi_\theta(a \mid s)$ and $\eta^\pi(s)$ is the expected number of visits to s, while $\nabla_\theta J = \sum_s w^\pi(s)\, \phi(s)$ with $w^\pi(s) = \sum_t \gamma^t \Pr_\pi(s_t = s)$. Both sum the same local directions $\phi(s)$ (the advantage form follows from Theorem 11.2), but the common update weights each state by its expected number of visits instead of its discounted number. For a *tabular* policy each state has its own parameters, so the common update multiplies the gradient's block for state s by the factor $\eta^\pi(s)/w^\pi(s) \geq 1$. It is then still an ascent direction, its inner product with $\nabla_\theta J$ being a sum of non-negative terms, and it vanishes exactly where the gradient does. With shared parameters the reweighting can turn the update around.

Example 11.5 Dropping γ^t can reverse the update

An agent visits two states, A then B, and the episode ends; it cannot tell them apart, so a single parameter controls both decisions, $p = \sigma(z) = \pi(\mathsf{R} \mid \mathsf{A}) = \pi(\mathsf{R} \mid \mathsf{B})$. In A, action R pays 1.5 and L pays 0; both lead to B. In B, L pays 2 and R pays 0. With $\gamma = 0.5$,

$$J(p) = 1.5\, p + \gamma \cdot 2(1-p) = 1 + 0.5\, p,$$

so the best policy in this class always plays R ($J = 1.5$) and the worst always plays L ($J = 1$). The gradient with respect to z sums the two states' contributions, $p(1-p) \times 1.5$ from A and $\gamma \times p(1-p) \times (-2)$ from B:

$$\frac{dJ}{dz} = p(1-p)\,(1.5 - 2\gamma) = 0.5\, p(1-p) > 0.$$

The common update weights B by 1 instead of γ and has expectation $p(1-p)(1.5-2) = -0.5\,p(1-p) < 0$. It drives p to 0: following it converges to the *worst* policy of the class.

♦

The example is small, but the phenomenon is general. Thomas (2014) showed that dropping γ^t biases actor-critic and natural actor-critic algorithms, and Nota and Thomas (2020) proved that the common update direction is, in general, not the gradient of any function, and constructed an example in which its fixed point is globally pessimal for both the discounted and the undiscounted objective. The most useful way to read the common practice is the one proposed with GAE (Schulman et al. 2016): in an episodic task, regard the *undiscounted* return as the real objective, and γ as a variance-reduction parameter that discounts the credit given for delayed effects. The common update is then a biased estimate of the undiscounted gradient, with a bias that shrinks as $\gamma \to 1$, and γ joins λ as a bias–variance dial.

PITFALL

Calling the γ^t-free REINFORCE or actor-critic update "an unbiased estimate of the policy gradient" is wrong. It is unbiased for neither the discounted nor the undiscounted objective. It usually works well, especially with γ close to 1, but its guarantees are those of a heuristic.

11.9.2 Average-reward policy gradients

In continuing tasks the natural objective is the long-run average reward per step, the *gain* $g(\theta) = \lim_{T\to\infty} \frac{1}{T}\mathbb{E}_{\pi_\theta}\big[\sum_{t<T} r_t\big]$ of Section 3.10.2. If every policy induces an ergodic Markov chain with stationary distribution μ_θ, the policy gradient theorem has an average-reward version (Sutton et al. 2000; Marbach and Tsitsiklis 2001):

$$\nabla_\theta g(\theta) = \sum_s \mu_\theta(s) \sum_a \nabla_\theta \pi_\theta(a \mid s)\, \tilde{Q}^{\pi_\theta}(s,a),$$

where $\tilde{Q}^\pi(s,a)$ is the *differential* action value, the expected sum of the rewards' deviations from the gain. The stationary distribution replaces the discounted visitation, and again its derivative does not appear. Baxter and Bartlett (2001) estimate this gradient from a single long trajectory, using a discount factor $\beta < 1$ inside the estimator as a bias–variance knob: the bias vanishes as $\beta \to 1$ while the variance grows. This is the clearest justification of the common practice above: in a continuing task, γ close to 1 is an approximation device for the average-reward gradient.

11.9.3 Score-function versus reparameterization gradients

The score-function estimator (11.4) treats the function being averaged as a black box. When that function is differentiable and the random variable can be written as a differentiable function of fixed noise, there is a second estimator. For a Gaussian action $a = \mu + \sigma\varepsilon$ with $\varepsilon \sim \mathcal{N}(0,1)$,

$$\frac{\partial}{\partial \mu}\,\mathbb{E}\big[f(a)\big] = \mathbb{E}\Big[f(\mu+\sigma\varepsilon)\,\frac{\varepsilon}{\sigma}\Big] \text{ (score function)} \;=\; \mathbb{E}\big[f'(\mu+\sigma\varepsilon)\big] \text{ (reparameterization).}$$

Both are unbiased, but their variances can differ enormously. For $f(a) = -(a-2)^2$, $\mu = 0$ and $\sigma = 1$, the true derivative is 4; the variance of a single sample is 87 for the score-function estimator, 42 with the baseline $b = \mathbb{E}f = -5$ (the analogue of a value baseline; the variance-minimizing constant of Proposition 11.3, $b^* = \mathbb{E}[\varepsilon^2 f] = -7$, gives 38), and only 4 for the reparameterization estimator (Exercise 11.13). The catch is that in reinforcement learning f is the action value $Q(s,a)$, which the agent does not know. Methods that learn a differentiable critic $Q_w(s,a)$ can use the reparameterization gradient through it: this is the idea behind the deterministic policy gradient and soft actor-critic (Theorem 13.1, Algorithm 13.3). They trade the score function's variance for the critic's bias. Mohamed et al. (2020) survey both families of estimators.

11.10 Compatible function approximation and natural actor-critic ●●●

Actor-critic methods replace Q^π by a learned approximation, which in general biases the gradient. Sutton et al. (2000) identified a class of critics for which it does not: critics that are linear in the policy's own score features.

Theorem 11.4 Compatible function approximation

Let $f_w(s,a) = w^\top \psi_\theta(s,a)$ with the *compatible features* $\psi_\theta(s,a) = \nabla_\theta \log \pi_\theta(a \mid s)$, and let w minimize the mean squared error $\mathbb{E}_{s\sim d^{\pi_\theta},\, a\sim\pi_\theta}\big[(Q^{\pi_\theta}(s,a) - f_w(s,a))^2\big]$. Then

$$\nabla_\theta J(\theta) = \frac{1}{1-\gamma}\, \mathbb{E}_{s\sim d^{\pi_\theta},\, a\sim\pi_\theta}\big[f_w(s,a)\, \nabla_\theta \log \pi_\theta(a \mid s)\big].$$

Proof. At a minimizer the gradient of the squared error with respect to w vanishes: $\mathbb{E}[(Q^{\pi_\theta}(s,a) - w^\top\psi_\theta(s,a))\,\psi_\theta(s,a)] = 0$, that is, $\mathbb{E}[Q^{\pi_\theta}\psi_\theta] = \mathbb{E}[f_w\psi_\theta]$. Multiply by $1/(1-\gamma)$ and use Theorem 11.1. ■

Three consequences make the theorem more than a curiosity. First, the critic only needs to be right *in the span of the score features*: errors orthogonal to them do not affect the gradient. Second, since the score has mean zero in every state, so does f_w; a compatible critic approximates the advantage, not Q, and any state-dependent part of Q is invisible to it. Third, the optimal weights are

$$w^* = \big(\mathbb{E}[\psi_\theta \psi_\theta^\top]\big)^{-1} \mathbb{E}[Q^{\pi_\theta}\psi_\theta] = (1-\gamma)\, F(\theta)^{-1} \nabla_\theta J(\theta),$$

where $F(\theta) = \mathbb{E}_{s\sim d^{\pi_\theta},a\sim\pi_\theta}[\psi_\theta\psi_\theta^\top]$ is the *Fisher information matrix* of the policy (assumed invertible). Up to the factor $1-\gamma$, the compatible critic's weight vector is the *natural gradient* $F^{-1}\nabla_\theta J$ (Kakade 2001), the subject of Section 12.4. On the Trail at the uniform policy, the compatible features are ± 0.5 in the coordinate of the current state, the best weights are $w^* = (1.52, 1.6)$, and f_{w^*} reproduces the advantages exactly, for example $f_{w^*}(\text{Start}, \text{go}) = 0.76 = 2.52 - 1.76$ (Exercise 11.15).

Natural actor-critic algorithms exploit this: they estimate the compatible weights w by a temporal-difference method and update the actor with $\theta \leftarrow \theta + \eta\, w$, a natural gradient step obtained without ever forming F (Peters and Schaal 2008a; Bhatnagar et al. 2009).

Do actor-critic methods converge? Konda and Tsitsiklis (2000) analyzed algorithms in which a linear critic, whose features span the compatible features, is trained by TD(λ) on a faster time scale than the actor, in the average-reward setting with ergodic policies. Their analysis shows convergence in the sense that the gradient norm becomes arbitrarily small infinitely often ($\liminf_k \|\nabla g(\theta_k)\| = 0$) when $\lambda = 1$, and comes within an error that vanishes as $\lambda \to 1$ otherwise. The two time scales are what make such results possible: the critic sees an almost stationary actor, and the actor sees an almost converged critic (Appendix B states the two-time-scale theorem). Off-policy actor-critic methods, which learn about the current policy from data of another, were developed by Degris et al. (2012) and are the starting point of Chapter 13.

RESEARCH NOTE

Open directions: finite-time guarantees for actor-critic methods with neural critics, where neither compatibility nor linearity holds; variance reduction beyond state-dependent baselines, where action-dependent control variates promised large gains that Tucker et al. (2018) found largely illusory on standard benchmarks; and a principled account of the γ^t-free updates used in practice. Chapter 14 covers what is known about the global convergence of policy gradient methods.

11.11 Summary

- Policy gradient methods parameterize the policy, $\pi_\theta(a \mid s)$, and improve it by stochastic gradient ascent on the expected return $J(\theta)$. They handle continuous actions and stochastic policies naturally and change behavior smoothly.
- Softmax policies (discrete actions) have the score $\mathbb{1}\{a = b\} - \pi(b \mid s)$ with respect to the logits; Gaussian policies have scores $(a - \mu)/\sigma^2$ and $(a - \mu)^2/\sigma^2 - 1$. On the Trail at the uniform policy, $\nabla_z J = (0.38, 0.18)$, a product of discounted visitation, sensitivity $p(1 - p)$ and an action-value difference in each state.
- The score-function trick and the policy gradient theorem express the gradient as an expectation of $\gamma^t Q(s_t, a_t) \nabla \log \pi_\theta(a_t \mid s_t)$ along the policy's own trajectories: no model and no derivative of the state distribution are needed, but the data must come from the current policy.
- REINFORCE replaces Q by the sampled return. It is unbiased and very noisy: on the Trail its single-episode estimates have total variance 1.67, against a gradient of length 0.42.
- Subtracting a state-dependent baseline never adds bias. The value baseline turns returns into advantages and cuts the Trail's variance by a factor of 17; the variance-minimizing baseline weights returns by squared scores (●●).
- Actor-critic methods bootstrap from a learned critic, trading variance for bias and allowing online learning; A2C runs many environments in parallel. GAE interpolates between one-step TD errors ($\lambda = 0$) and Monte Carlo advantages ($\lambda = 1$), and the best λ grows with the batch size.
- Time-limit truncations must bootstrap from the value of the final state; treating them as terminations is a common and silent bug.
- Most implementations drop γ^t; the resulting update is not the gradient of any objective and can, with shared parameters, point downhill (●●). Compatible critics give exact gradients and connect actor-critic methods to natural gradients (●●●).

Plain gradient steps measure distance in parameter space, and Chapter 12 shows why that is the wrong ruler: it replaces it with distances between policies, leading to natural gradients, trust regions and PPO.

11.12 Further Reading

Sutton and Barto (2018, ch. 13) give a gentle treatment of REINFORCE, baselines and actor-critic methods, including the episodic algorithms with the γ^t factor used here. The original REINFORCE paper, Williams (1992), is still worth reading for its general view of stochastic units and baselines. The policy gradient theorem with function approximation and the compatibility result are in Sutton et al. (2000); Marbach and Tsitsiklis (2001) and Baxter and Bartlett (2001) develop the average-reward versions and single-trajectory estimators. Greensmith et al. (2004) is the definitive analysis of baselines and critics as control variates: it derives the variance-minimizing baseline and shows that neither the average reward as a baseline nor the true value function as a critic is optimal in general. Weaver and Tao (2001) study the optimal constant baseline.

Peters and Schaal (2008b) survey policy gradient methods for robotics, from finite differences to natural actor-critic, with careful attention to baselines; Peters and Schaal (2008a) and Bhatnagar et al. (2009) develop natural actor-critic algorithms. For deep actor-critic methods, read Mnih et al. (2016) (A3C) and Schulman et al. (2016) (GAE), whose experiments also show how γ and λ interact. Mohamed et al. (2020) is an excellent survey of score-function and reparameterization gradient estimators beyond reinforcement learning.

On the convergence of actor-critic methods, Konda and Tsitsiklis (2000) is the classic two-time-scale analysis, and Degris et al. (2012) extends actor-critic to off-policy data. For the theory of policy gradient methods in tabular problems, including global convergence and the plateaus of Figure 11.1, see A. Agarwal et al. (2021) and Chapter 14.

(●●●) The discounting subtlety is analyzed by Thomas (2014) and Nota and Thomas (2020). On variance reduction, Wu et al. (2018) propose action-dependent baselines and Tucker et al. (2018) re-examine their benefits. Ahmed et al. (2019) study the effect of entropy regularization on the optimization landscape.

11.13 Exercises

Exercise 11.1 ● The Trail's gradient at another policy

Take $p_0 = 0.8$ and $p_1 = 0.5$ (that is, $z_0 = \ln 4$ and $z_1 = 0$) and $\gamma = 0.9$. Compute J, the action values in Start and Middle, and $\nabla_z J$, using the three-factor table of Example 11.1. Which component is now larger, and why did the ranking change compared with the uniform policy?

Exercise 11.2 ● Score functions

(a) Derive (11.2) for a softmax policy. (b) Show that for linear preferences $h_\theta(s,a) = \theta^\top x(s,a)$ the score is $x(s,a) - \sum_b \pi_\theta(b \mid s)\, x(s,b)$. (c) Derive the two scores of a Gaussian policy

given in Section 11.2.3. (d) Verify directly that both Gaussian scores have mean zero when $a \sim \mathcal{N}(\mu, \sigma^2)$.

Exercise 11.3 • One REINFORCE update by hand

At the uniform Trail policy ($\gamma = 0.9$) the agent observes the episode go, exit. (a) Compute the REINFORCE estimate $\hat{g}$. (b) Take one step $z \leftarrow z + \eta\hat{g}$ with $\eta = 0.5$, and give the new p_0, p_1 and J. (c) The step lowered p_1, although going on from Middle is better than exiting. Explain why, then repeat (a) and (b) with the baseline $b(\text{Start}) = 1.76$, $b(\text{Middle}) = 2.8$.

Exercise 11.4 • A useless baseline and an all-positive reward

(a) Verify Theorem 11.2 directly for one state with two actions and a logistic policy $p = \sigma(z)$. (b) Recompute the three estimates of Example 11.2 with the baseline $b \equiv 10$ in every state; check their mean and compute their total variance. (c) In CartPole every reward is $+1$. Explain in a paragraph what REINFORCE without a baseline does to the probability of each action it takes, and what a learned value baseline changes.

Exercise 11.5 • The two ends of GAE

Using only the definition $\delta_k = r_k + \gamma V(s_{k+1}) - V(s_k)$, show that for an episode that terminates at time T, $\sum_{l=0}^{T-t-1} \gamma^l \delta_{t+l} = G_t - V(s_t)$, and conclude that GAE with $\lambda = 1$ is REINFORCE's advantage with a value baseline. What is GAE with $\lambda = 0$, and which algorithm of this chapter uses it?

Exercise 11.6 • GAE with a truncation

A rollout of one environment has four steps with rewards $(1, 0, 2, 1)$ and critic values $V(s_0), \dots, V(s_3) = (2, 3, 1, 0.5)$. After step 1 the episode is truncated by a time limit in a state whose value is 1.5, and a new episode starts in s_2. The rollout ends after step 3, in the middle of that episode, in a state with value 2. With $\gamma = 0.9$ and $\lambda = 0.8$: (a) compute the four TD errors and the four GAE advantages; (b) recompute them as a buggy implementation would that treats the truncation as a termination; (c) recompute them as an implementation would that forgets to stop the sum at the truncation. Which advantages change in each case?

Exercise 11.7 • The entropy bonus

(a) For a two-action logistic policy $p = \sigma(z)$, show that the entropy $\mathcal{H}(p) = -p \log p - (1 - p) \log(1 - p)$ has derivative $d\mathcal{H}/dz = -p(1 - p)\, z$. For which $z \geq 0$ is the pull toward $z = 0$ strongest? (b) Programming: in Lab 11.1, train A2C on CartPole for 100,000 steps with $c_H \in \{0, 0.01, 0.1\}$ (three seeds each) and report the final returns and policy entropies. What does a large coefficient do?

Exercise 11.8 ●● The optimal baseline for two actions

(a) Show that for a state with two actions and a logistic policy $p = \pi(\text{go} \mid s) = \sigma(z)$, the variance-minimizing constant baseline of Proposition 11.3 for the score with respect to z is $b^* = (1-p)\,Q(s,\text{go}) + p\,Q(s,\text{exit})$, whereas $V(s) = p\,Q(s,\text{go}) + (1-p)\,Q(s,\text{exit})$. Explain the swapped weights. (b) On the Trail with $p_0 = 0.8$, $p_1 = 0.5$, compute b^* for Start and the variance of the z_0 component of the REINFORCE estimate with no baseline, with $b = V(\text{Start})$ and with $b = b^*$.

Exercise 11.9 ●● When does reward-to-go help?

(a) Show that on the Trail, replacing $\gamma^t G_t$ by the full return G_0 in every term of the REINFORCE estimate gives exactly the same estimates. (b) Modify the Trail so that go in Start pays 0.5. At the uniform policy with $\gamma = 0.9$, compute the distribution, mean and total variance of both estimators. (c) Explain, using Step 1 of the trajectory proof in Section 11.3.3, why the two estimators always have the same mean.

Exercise 11.10 ●● The gradient of every value

For the Trail at the uniform policy, compute the local directions $\phi(s) = \sum_a \nabla_z \pi(a \mid s)\, Q(s,a)$ for Start, Middle and Goal, and solve the gradient's Bellman equation of Section 11.3.4 to find $\nabla_z V(\text{Middle})$ and $\nabla_z V(\text{Start})$. Check the second against Example 11.1.

Exercise 11.11 ●● A critic that starts at zero

One-step actor-critic (Algorithm 11.2) runs on the Trail with a tabular critic initialized to $V_w \equiv 0$, from the uniform policy. (a) At the first visit to Start, compute the expected actor update $\mathbb{E}[\delta\, \nabla_{z_0} \log \pi(a \mid \text{Start})]$ and compare its sign with the true gradient. (b) Compute the same expectation when the critic equals V^π of the uniform policy. (c) Explain why the bias of (a) is temporary in this problem, and how the critic's initialization and step size affect how long it lasts.

Exercise 11.12 ●● When does dropping γ^t reverse the update?

In the aliased example of Example 11.5, keep the rewards but let $\gamma \in [0,1)$ be arbitrary. (a) Find the true gradient dJ/dz and the expectation of the common update as functions of p and γ, and the range of γ in which their signs disagree. (b) Which policy is optimal for each γ, and where does the common update converge? (c) Give each state its own parameter and show that the two directions then always agree in sign, component by component.

Exercise 11.13 ●● Score function versus reparameterization

Let $a = \mu + \sigma\varepsilon$ with $\varepsilon \sim \mathcal{N}(0,1)$, $\mu = 0$, $\sigma = 1$, and $f(a) = -(a-2)^2$. Using $\mathbb{E}[\varepsilon^2] = 1$, $\mathbb{E}[\varepsilon^4] = 3$ and $\mathbb{E}[\varepsilon^6] = 15$, compute the mean and variance of (a) the score-function estimator $f(a)\,\varepsilon$, (b) the same with the baseline $b = \mathbb{E}[f(a)]$, and (c) the reparameterization estimator $f'(a)$.

Exercise 11.14 ●● GAE as a λ-return

(a) Show that $\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = G_t^\lambda - V(s_t)$, where $G_t^\lambda = (1-\lambda)\sum_{n\geq 1}\lambda^{n-1}G_{t:t+n}$ and the n-step returns bootstrap from V. (b) Show that $\mathbb{E}[\hat{A}_t^{\mathrm{GAE}(\gamma,1)} \mid s_t, a_t] = Q^\pi(s_t,a_t) - V(s_t)$ for any critic V (with $V = 0$ at terminal states), and conclude that GAE with $\lambda = 1$ gives an unbiased policy gradient estimate however poor the critic. (c) Why does the argument fail for $\lambda < 1$?

Exercise 11.15 ●●● A compatible critic for the Trail

(a) Write down the compatible features $\psi(s,a)$ of the Trail's logit policy at the uniform policy, for all four state-action pairs of Start and Middle. (b) Find the weights w^* that minimize $\mathbb{E}_{s\sim d^\pi, a\sim\pi}[(Q^\pi(s,a) - w^\top\psi(s,a))^2]$ and show that $f_{w^*} = A^\pi$ on these pairs. (c) Compute the Fisher matrix F and verify $w^* = (1-\gamma)F^{-1}\nabla_z J$. (d) Why can a compatible critic represent A^π but not Q^π?

Exercise 11.16 ●●● Beyond state-dependent baselines

State-dependent baselines leave a large part of the variance untouched. Read Wu et al. (2018) and Tucker et al. (2018). Explain how an action-dependent baseline can be used without introducing bias, summarize the variance decomposition that Tucker et al. (2018) use, and say what they concluded about the reported gains. Propose one experiment on a problem from this book's labs that would test their conclusion.

Exercise 11.17 ●●● What do discounted policy gradient methods optimize?

Read Nota and Thomas (2020). Summarize their argument that the common update is not a gradient, and describe their counterexample. Then relate it to Example 11.5 and to the view of γ as a variance-reduction parameter (Schulman et al. 2016). Under which conditions would you expect the common update to be harmless, and how could you check this numerically for a given MDP and policy class?

CHAPTER 12

Natural Gradients, Trust Regions, and PPO

WHAT YOU WILL LEARN

- Explain why a fixed step size in parameter space is fragile for policy gradient methods, and why the size of a step should be measured by how much the policy changes.
- Define the surrogate objective, compute it on the Trail, and show that it matches the true objective to first order.
- Compute the Fisher information matrix and the natural gradient of the Trail's policy, and explain why natural gradients do not get stuck on plateaus.
- Describe TRPO: the KL-constrained problem, its solution by conjugate gradient with Fisher-vector products, and the line search.
- Explain PPO's clipped objective case by case, implement PPO with GAE, minibatches and epochs, and read its diagnostics.
- Prove the monotonic improvement bound and the parameterization invariance of the natural gradient (●●).
- Discuss what PPO's clipping does and does not guarantee, and its relation to mirror descent (●●●).

The policy gradient methods of Chapter 11 tell an agent which way to change its parameters. They do not say how far. That second question turns out to be the hard one. A step that is too small wastes data, which in reinforcement learning is expensive, because every sample comes from interacting with the environment. A step that is too large can wreck the policy: it moves into a region where the agent behaves badly, collects bad data and never recovers. Worse, the same step size can be too small in one part of parameter space and too large in another, because a unit of parameter change can mean a tiny change in behavior or a drastic one.

This chapter is about measuring steps in the right units. The right unit is the change in the *policy*, measured by the Kullback–Leibler (KL) divergence between the old and the new action distributions. Three ideas follow from it. The *surrogate objective* predicts how good a new policy will be using only data from the old one, and the *monotonic improvement theorem* says how far that prediction can be trusted. The *natural policy gradient* is steepest ascent when distance is measured in KL rather than in parameters. And *trust-region methods* turn these ideas into algorithms: TRPO enforces a KL constraint with second-order machinery, and PPO approximates it with a simple clipped objective that can be optimized by minibatch gradient descent. PPO is today the default on-policy algorithm in deep reinforcement learning and the workhorse of reinforcement learning from human feedback (Chapter 22).

Section 12.1 shows the step-size problem on the Trail. Section 12.2 introduces surrogate objectives, and Section 12.3 (●●) proves the bound that justifies trusting them within a region. Section 12.4 develops the natural policy gradient, with a ●● subsection on why the Fisher matrix measures KL divergence. Section 12.5 presents TRPO and Section 12.6 PPO; Section 12.7 covers the implementation details and diagnostics that make PPO work, with experiments on CartPole and Pendulum. Section 12.8 (●●●) asks what PPO actually optimizes. On a first reading, skip the two higher-level sections; everything else is self-contained, given Chapter 11.

12.1 The step-size problem

Policy gradient ascent updates $\theta \leftarrow \theta + \eta\,\hat{g}$, where $\hat{g}$ estimates $\nabla_\theta J(\theta)$ and η is a step size. In supervised learning a poorly chosen step size costs time. In reinforcement learning it can cost the whole run, for three reasons.

The same parameter step can mean very different policy changes. To compare two action distributions we use the *Kullback–Leibler divergence*, $D_{\mathrm{KL}}(p \,\|\, q) = \sum_a p(a) \log \frac{p(a)}{q(a)}$, which is zero when $p = q$ and positive otherwise (Section A.6). For a two-action policy with logit z, so that the probability of the first action is $p = \sigma(z)$, it is

$$D_{\mathrm{KL}}\big(\sigma(z) \,\|\, \sigma(z')\big) = \sigma(z) \log \frac{\sigma(z)}{\sigma(z')} + \big(1 - \sigma(z)\big) \log \frac{1 - \sigma(z)}{1 - \sigma(z')}.$$

Now take the same step, $z \to z + 1$, from two places. From $z = 0$ the probability moves from 0.5 to 0.731 and the KL divergence is 0.120. From $z = 4$ it moves from 0.982 to 0.993 and the KL divergence is 0.0066, eighteen times smaller (Figure 12.1a). A step size that is safe near $z = 4$ is reckless near $z = 0$, and one that is safe near $z = 0$ crawls near $z = 4$.

Noisy steps can collapse the policy. Policy gradient estimates are noisy, and a large step taken in a bad direction can push the policy into a nearly deterministic region. There the gradient nearly vanishes, as we saw on the Trail in Chapter 11, and the policy is stuck. On the Trail, REINFORCE with one episode per update illustrates this vividly. If the first episode exits at Start, which happens half the time, the estimate is $(-0.5, 0)$; with $\eta = 10$, one step sets $z_0 = -5$, and the hiker now exits 99.3% of the time. Each further exit nudges z_0 down only a little more, and the hiker goes on, and so learns that going on pays, only once every 150 episodes or so. Figure 12.1b counts the outcomes of 200 runs of 300 episodes for each step size. With $\eta = 0.3$ every run finds the optimal policy. With $\eta = 3$ only 62% do; with $\eta = 10$, 27%, while 45.5% are stuck exiting at Start and 27.5% are stuck exiting at Middle.

A bad step changes the data. In supervised learning a bad step is corrected by the next batch, which comes from the same fixed data set. In reinforcement learning the next batch comes from the new policy. A policy that has learned to exit at once never visits Middle again, so it never collects the data that would show it its mistake. Bad steps are self-reinforcing.

All three problems have the same cure: control the step by how much it changes the *policy*, not the parameters. The rest of the chapter develops the two tools needed for this, a way of predicting the effect of a policy change before making it, and a way of measuring and limiting the change.

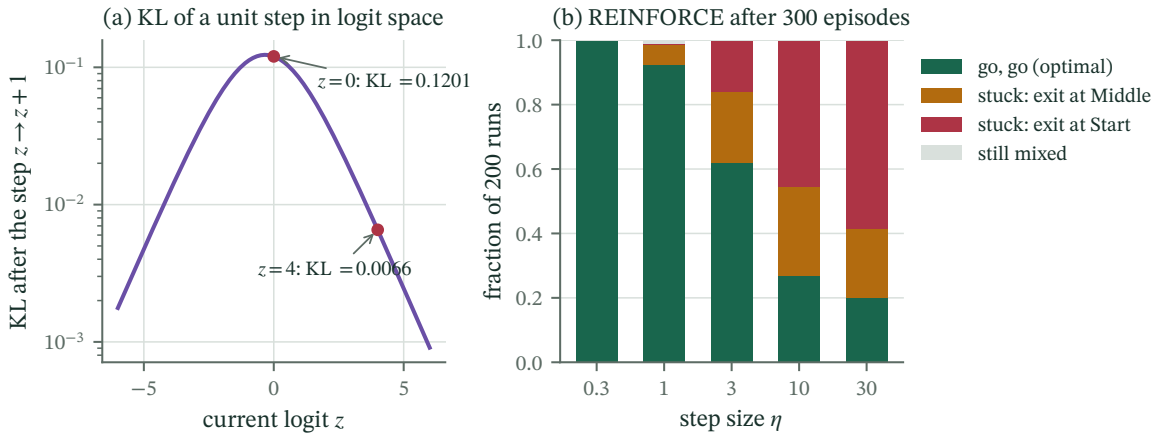


Figure 12.1. The step-size problem on the Trail. (a) The KL divergence between the policies before and after the same unit step in logit space, $z \to z+1$, as a function of the starting logit (log scale): the step is eighteen times larger in policy terms at $z=0$ than at $z=4$. (b) REINFORCE on the Trail with one episode per update, for five step sizes (200 runs of 300 episodes each). Large steps leave most runs stuck in a bad deterministic policy from which the vanishing gradient never rescues them.

KEY IDEA

Measure the size of a policy update by how much it changes the action distributions, for example by their KL divergence, not by the length of the parameter step. Steps of a fixed size in policy space are safe everywhere; steps of a fixed size in parameter space are not.

12.2 Surrogate objectives

Suppose we have collected data with a policy π_{old} and are considering a new policy π. How good will π be? The *performance difference lemma* of Chapter 3 (Lemma 3.10) gives an exact answer:

$$J(\pi) - J(\pi_{\text{old}}) = \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d^{\pi},\,a\sim\pi(\cdot|s)}\big[A^{\pi_{\text{old}}}(s,a)\big].$$

In words: the new policy's gain is its average advantage, measured with the old policy's advantage function $A^{\pi_{\text{old}}} = Q^{\pi_{\text{old}}} - V^{\pi_{\text{old}}}$, over the states the *new* policy visits. Here d^{π} is the discounted state-visitation distribution of Definition 3.8. The advantages can be estimated from the old data, but the states cannot: d^{π} is the distribution of a policy we have not run yet. The *surrogate objective* simply replaces the new policy's state distribution by the old one's:

$$L_{\pi_{\text{old}}}(\pi) = J(\pi_{\text{old}}) + \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d^{\pi_{\text{old}}},\,a\sim\pi(\cdot|s)}\big[A^{\pi_{\text{old}}}(s,a)\big]. \tag{12.1}$$

The states now come from the old policy, but the actions still come from the new one. One more step makes the surrogate computable from old data alone. Writing the inner expectation over actions as an expectation under π_{old}, reweighted by the *probability ratio*

$$\rho(\theta) = \frac{\pi_{\theta}(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)},$$

gives the importance-sampled form, for a parameterized policy π_θ:

$$L_{\theta_{\text{old}}}(\theta) = J(\theta_{\text{old}}) + \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d^{\pi_{\theta_{\text{old}}}},\,a\sim\pi_{\theta_{\text{old}}}(\cdot|s)}\big[\rho(\theta)\,A^{\pi_{\theta_{\text{old}}}}(s,a)\big]. \tag{12.2}$$

Given a batch of transitions (s_t, a_t) collected with $\pi_{\theta_{\text{old}}}$ and advantage estimates $\hat{A}_t$, the sample version is the batch average $\frac{1}{N}\sum_t \rho_t(\theta)\hat{A}_t$, up to a constant and a positive factor, where $\rho_t(\theta) = \pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$. This is what TRPO and PPO optimize. Its key property is that it can be evaluated, and differentiated, for *any* θ using the *same* batch, so one batch can support many gradient steps.

How good a stand-in is L for J? Near θ_{old}, perfect to first order.

Proposition 12.1 The surrogate matches the objective to first order

For a differentiable policy π_θ, $L_{\theta_{\text{old}}}(\theta_{\text{old}}) = J(\theta_{\text{old}})$ and $\nabla_\theta L_{\theta_{\text{old}}}(\theta)\big|_{\theta=\theta_{\text{old}}} = \nabla_\theta J(\theta)\big|_{\theta=\theta_{\text{old}}}$.

Proof. At $\theta = \theta_{\text{old}}$ every ratio is 1, and $\mathbb{E}_{a\sim\pi_{\text{old}}}[A^{\pi_{\text{old}}}(s,a)] = 0$ in every state, so $L = J(\theta_{\text{old}})$. The gradient of the ratio at θ_{old} is $\nabla_\theta \pi_\theta(a \mid s)/\pi_{\theta_{\text{old}}}(a \mid s) = \nabla_\theta \log \pi_\theta(a \mid s)$, so

$$\nabla_\theta L\big|_{\theta_{\text{old}}} = \frac{1}{1-\gamma}\,\mathbb{E}_{s\sim d^{\pi_{\text{old}}},\,a\sim\pi_{\text{old}}}\big[A^{\pi_{\text{old}}}(s,a)\,\nabla_\theta \log \pi_\theta(a \mid s)\big],$$

which is the policy gradient theorem (Theorem 11.1) with the value baseline subtracted (Theorem 11.2). ∎

The proposition says that following the gradient of the surrogate, from the old parameters, is the same as following the policy gradient. The difference appears when we take larger steps: J is curved in ways L does not see, because L ignores how the new policy changes the states it visits.

The surrogate on the Trail. Everything can be computed exactly. Let the old policy be (q_0, q_1) and the new one (p_0, p_1), and write $\Delta Q_s = Q^{\pi_{\text{old}}}(s, \text{go}) - Q^{\pi_{\text{old}}}(s, \text{exit})$. The old policy visits Start with discounted weight 1 and Middle with weight γq_0; in each state the new policy's average advantage is $(p - q)\,\Delta Q_s$, because $A^{\pi_{\text{old}}}(s, \text{go}) = (1-q)\Delta Q_s$ and $A^{\pi_{\text{old}}}(s, \text{exit}) = -q\,\Delta Q_s$. So

$$L = J(\pi_{\text{old}}) + (p_0 - q_0)\,\Delta Q_{\text{Start}} + \gamma\, q_0\,(p_1 - q_1)\,\Delta Q_{\text{Middle}}.$$

The surrogate is *linear* in the new probabilities. The true objective is not: expanding Chapter 1's formula $J = (1-p_0) + 2\gamma p_0(1-p_1) + 4\gamma^2 p_0 p_1$ around the old policy gives

$$J(\pi) - L_{\pi_{\text{old}}}(\pi) = \gamma\,\Delta Q_{\text{Middle}}\,(p_0 - q_0)(p_1 - q_1).$$

The error is the product of the two policy changes: second order, as Proposition 12.1 promised. Its origin is visible: when the new policy changes p_0, it changes how often Middle is visited, and the surrogate, which keeps the old visitation, misses the interaction. When both probabilities move the same way the surrogate is pessimistic; when they move in opposite directions it is optimistic, which is more dangerous.

Figure 12.2a follows both objectives along a line through parameter space, starting at the uniform policy and moving in the gradient direction. For short steps L and J coincide. A step of length

0.5 (which changes the policy by a maximum KL of 0.025) gives $J = 1.976$ and $L = 1.967$; a step of length 2 (maximum KL 0.36) gives $J = 2.556$ and $L = 2.451$. The figure also shows a lower bound on J that the next section proves; it guarantees improvement only for very short steps.

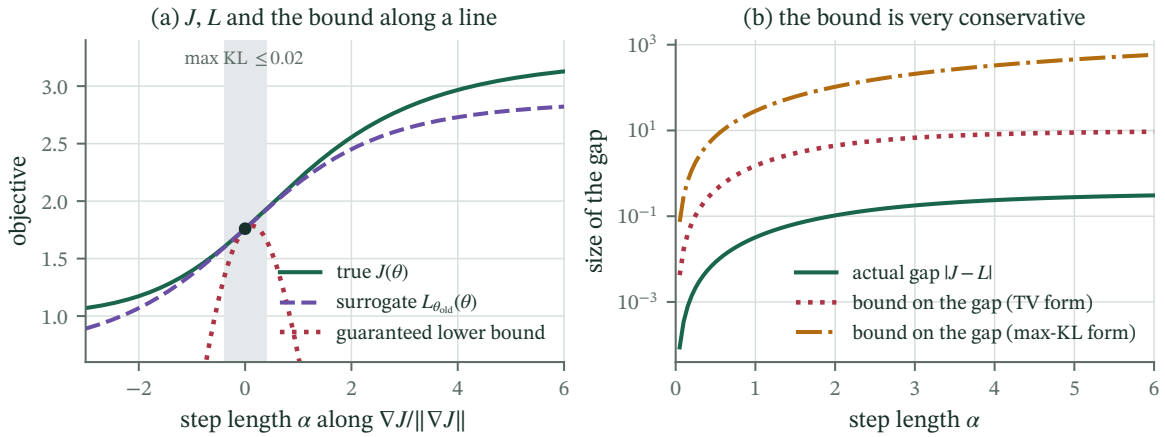


Figure 12.2. The surrogate on the Trail ($\gamma = 0.9$). (a) The true objective J and the surrogate L along a line through the uniform policy in the direction of the gradient (α is the distance in logit space). They agree near $\alpha = 0$ and drift apart for long steps. The dotted curve is the guaranteed lower bound of Theorem 12.3 (total-variation form); the shaded band marks the steps whose maximum KL divergence is at most 0.02. (b) The actual gap $|J-L|$ and the theorem's two bounds on it (log scale): the bounds hold, but they are one to three orders of magnitude too large.

12.3 The monotonic improvement bound ●●

How far can the surrogate be trusted? The answer is a bound on $|J - L|$ in terms of how much the policy changes. We measure the change in state s by the *total variation distance* $D_{\mathrm{TV}}\big(\pi'(\cdot \mid s), \pi(\cdot \mid s)\big) = \frac{1}{2}\sum_a |\pi'(a \mid s) - \pi(a \mid s)|$, and first bound how much the state distribution can move.

Lemma 12.2 Policy changes move the state distribution by a bounded amount

For any two policies π and π' of a finite MDP with $\gamma < 1$,

$$\left\| d^{\pi'} - d^{\pi} \right\|_1 \le \frac{2\gamma}{1-\gamma}\, \mathbb{E}_{s\sim d^{\pi}}\Big[D_{\mathrm{TV}}\big(\pi'(\cdot \mid s), \pi(\cdot \mid s)\big)\Big].$$

Proof. Treat distributions as row vectors. With $P_\pi(s,s') = \sum_a \pi(a \mid s) P(s' \mid s,a)$ and $G_\pi = (I - \gamma P_\pi)^{-1} = \sum_{t\ge 0} \gamma^t P_\pi^t$, the visitation distribution is $d^\pi = (1-\gamma)\, d_0^{\mathsf{T}} G_\pi$. The two inverses satisfy $G_{\pi'} - G_\pi = G_\pi\big(G_\pi^{-1} - G_{\pi'}^{-1}\big)G_{\pi'} = \gamma\, G_\pi(P_{\pi'} - P_\pi)G_{\pi'}$, so

$$d^{\pi'} - d^{\pi} = (1-\gamma)\, d_0^{\mathsf{T}}(G_{\pi'} - G_\pi) = \gamma\, (d^{\pi})^{\mathsf{T}}(P_{\pi'} - P_\pi)\, G_{\pi'}.$$

For any row vector x, $\|x^{\mathsf{T}} G_{\pi'}\|_1 \le \sum_t \gamma^t \|x^{\mathsf{T}} P_{\pi'}^t\|_1 \le \|x\|_1/(1-\gamma)$, because multiplying by a

stochastic matrix does not increase the ℓ_1 norm. And

$$\begin{aligned}\big\| (d^{\pi})^{\top}(P_{\pi'} - P_{\pi}) \big\|_1 &= \sum_{s'} \Big| \sum_{s} d^{\pi}(s) \sum_{a} \big(\pi'(a \mid s) - \pi(a \mid s)\big) P(s' \mid s, a) \Big| \\ &\le \sum_{s} d^{\pi}(s) \sum_{a} |\pi'(a \mid s) - \pi(a \mid s)|,\end{aligned}$$

which is $2\,\mathbb{E}_{s\sim d^{\pi}}[D_{\mathrm{TV}}(\pi'(\cdot \mid s), \pi(\cdot \mid s))]$. Combining the three displays proves the lemma. ∎

Theorem 12.3 Monotonic improvement bound

Let π and π' be policies of a finite MDP with $\gamma < 1$, let $\bar{A}(s) = \mathbb{E}_{a\sim\pi'(\cdot|s)}[A^{\pi}(s,a)]$, $\bar{\epsilon} = \max_s |\bar{A}(s)|$ and $\epsilon_A = \max_{s,a} |A^{\pi}(s,a)|$. Then

$$J(\pi') \ge L_{\pi}(\pi') - \frac{2\gamma\,\bar{\epsilon}}{(1-\gamma)^2}\, \mathbb{E}_{s\sim d^{\pi}}\Big[D_{\mathrm{TV}}\big(\pi'(\cdot \mid s), \pi(\cdot \mid s)\big) \Big],$$

and, as a consequence,

$$J(\pi') \ge L_{\pi}(\pi') - \frac{4\gamma\,\epsilon_A}{(1-\gamma)^2}\, \max_{s} D_{\mathrm{KL}}\big(\pi(\cdot \mid s) \,\|\, \pi'(\cdot \mid s)\big).$$

The first form is due to Achiam et al. (2017), the second to Schulman et al. (2015), who built on the analysis of conservative policy iteration by Kakade and Langford (2002).

Proof. By the performance difference lemma and the definition (12.1),

$$J(\pi') - L_{\pi}(\pi') = \frac{1}{1-\gamma} \sum_{s} \big(d^{\pi'}(s) - d^{\pi}(s)\big)\, \bar{A}(s) \ge -\frac{\bar{\epsilon}}{1-\gamma} \left\| d^{\pi'} - d^{\pi} \right\|_1,$$

and Lemma 12.2 gives the first inequality. For the second, write $\alpha = \max_s D_{\mathrm{TV}}(\pi'(\cdot \mid s), \pi(\cdot \mid s))$. Since $\mathbb{E}_{a\sim\pi}[A^{\pi}(s,a)] = 0$, $\bar{A}(s) = \sum_a \big(\pi'(a \mid s) - \pi(a \mid s)\big) A^{\pi}(s,a)$, so $|\bar{A}(s)| \le 2\, D_{\mathrm{TV}}(\pi'(\cdot \mid s), \pi(\cdot \mid s))\, \epsilon_A \le 2\alpha\epsilon_A$; and $\mathbb{E}_{s\sim d^{\pi}}[D_{\mathrm{TV}}] \le \alpha$. The penalty of the first inequality is therefore at most $4\gamma\epsilon_A\alpha^2/(1-\gamma)^2$. Finally, Pinsker's inequality $D_{\mathrm{TV}}(p,q) \le \sqrt{D_{\mathrm{KL}}(p\,\|\,q)/2}$ (Section A.6) gives $\alpha^2 \le \tfrac{1}{2} \max_s D_{\mathrm{KL}}(\pi(\cdot \mid s) \,\|\, \pi'(\cdot \mid s)) \le \max_s D_{\mathrm{KL}}(\pi(\cdot \mid s) \,\|\, \pi'(\cdot \mid s))$. ∎

The theorem turns the surrogate into a *lower bound* that is tight at the current policy, and this yields an algorithm that can only improve.

Corollary 12.4 Minorize–maximize policy iteration improves monotonically

Let $C = 4\gamma\epsilon_A/(1-\gamma)^2$ be computed for the current policy π_k, and let $M_k(\pi) = L_{\pi_k}(\pi) - C \max_s D_{\mathrm{KL}}(\pi_k(\cdot \mid s) \,\|\, \pi(\cdot \mid s))$. If $M_k(\pi_{k+1}) \ge M_k(\pi_k)$, in particular if π_{k+1} maximizes M_k, then $J(\pi_{k+1}) \ge J(\pi_k)$.

Proof. $J(\pi_{k+1}) \ge M_k(\pi_{k+1}) \ge M_k(\pi_k) = L_{\pi_k}(\pi_k) = J(\pi_k)$, using Theorem 12.3, the assumption, and the fact that the KL term vanishes at $\pi = \pi_k$. ∎

This is a *minorize–maximize* scheme: at every iteration, maximize a function that lies below J and touches it at the current point. Kakade and Langford (2002) had earlier proved a similar guarantee for *conservative policy iteration*, which moves to a mixture $(1-\alpha)\pi_k + \alpha\pi'$ of the current policy and a greedy one, with α small enough for the improvement to outweigh the penalty; the theorem above extends the idea to arbitrary policies.

As a practical method the bound is far too conservative. On the Trail at the uniform policy, $\epsilon_A = 0.8$ and $C = 288$; along the line of Figure 12.2, the total-variation form guarantees an improvement only for steps shorter than about 0.25, and its best guarantee is an improvement of about 0.025 (to $J \geq 1.785$), for a step of length about 0.12 that actually improves J by about 0.05. At length 0.5 the true gap $|J - L|$ is 0.0085, while the two bounds on it are 0.41 and 7.3. The factor $1/(1-\gamma)^2$ accounts for policy changes compounding over an effective horizon of $1/(1-\gamma) = 10$ steps, but Trail episodes last at most three. Practical algorithms therefore keep the *shape* of the theorem, a surrogate plus a limit on the KL divergence, and choose the limit by hand (Section 12.5). They also replace exact advantages by estimates and the maximum over states by an average, so the guarantee of Corollary 12.4 no longer applies to them.

12.4 Natural policy gradient

The surrogate tells us what a policy change is worth; we now need a way to choose the change. Plain gradient ascent takes the steepest step for a given Euclidean length in parameter space: among all steps $\Delta\theta$ with $\|\Delta\theta\| \leq r$, the one that increases the linear approximation $J(\theta) + g^{\top}\Delta\theta$ the most is $r\, g/\|g\|$, with $g = \nabla_\theta J(\theta)$. Section 12.1 showed that Euclidean length is the wrong ruler. The *natural gradient* uses the right one.

KL divergence for small steps. For a small step $\Delta\theta$, the average KL divergence between the old and new action distributions is, to second order, a quadratic form:

$$\mathbb{E}_{s\sim d^{\pi_\theta}}\Big[D_{\mathrm{KL}}\big(\pi_\theta(\cdot \mid s) \,\|\, \pi_{\theta+\Delta\theta}(\cdot \mid s)\big)\Big] \approx \tfrac{1}{2}\,\Delta\theta^{\top} F(\theta)\,\Delta\theta,$$

where $F(\theta)$ is the *Fisher information matrix* of the policy,

$$F(\theta) = \mathbb{E}_{s\sim d^{\pi_\theta},\, a\sim\pi_\theta(\cdot|s)}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)^{\top}\Big]. \tag{12.3}$$

F is the covariance matrix of the score; it is symmetric and positive semidefinite. Section 12.4.1 proves the approximation. For a single two-action decision with logit z the Fisher information is the number $\sigma(z)(1-\sigma(z))$, so a step Δz costs about $\tfrac{1}{2}\sigma(z)(1-\sigma(z))\,\Delta z^2$ in KL: 0.125 for a unit step at $z = 0$ (the exact value is 0.120) and 0.0088 at $z = 4$ (exact: 0.0066). The Fisher matrix is the local exchange rate between parameter changes and policy changes.

Steepest ascent in KL. Now choose the step that increases the linear approximation the most among steps of a given KL size δ:

$$\max_{\Delta\theta}\; g^{\top}\Delta\theta \quad \text{subject to} \quad \tfrac{1}{2}\,\Delta\theta^{\top} F \Delta\theta \leq \delta.$$

With a Lagrange multiplier μ, stationarity of $g^{\top}\Delta\theta - \mu(\tfrac{1}{2}\Delta\theta^{\top}F\Delta\theta - \delta)$ gives $g = \mu F \Delta\theta$, so

$\Delta\theta = F^{-1}g/\mu$, and making the constraint active fixes μ:

$$\Delta\theta^* = \sqrt{\frac{2\delta}{g^\top F^{-1} g}}\, F^{-1} g. \tag{12.4}$$

The direction $\tilde{\nabla}_\theta J = F(\theta)^{-1}\nabla_\theta J(\theta)$ is the *natural gradient* (Amari 1998); Kakade (2001) introduced it for policies. (If F is singular, as for softmax policies, whose logits can all be shifted by a constant without changing the policy, the pseudo-inverse F^+ takes its place.) Natural gradient ascent with step size η updates $\theta \leftarrow \theta + \eta\, F(\theta)^{-1}\nabla_\theta J(\theta)$.

The Trail. At the uniform policy the scores are ± 0.5 in the coordinate of the current state, the discounted visitation is $d^\pi(\text{Start}) = 1-\gamma = 0.1$ and $d^\pi(\text{Middle}) = (1-\gamma)\gamma p_0 = 0.045$, and in each state the score has variance $p(1-p) = 0.25$. So

$$F = \begin{pmatrix} (1-\gamma)\, p_0(1-p_0) & 0 \\ 0 & (1-\gamma)\gamma\, p_0\, p_1(1-p_1) \end{pmatrix} = \begin{pmatrix} 0.025 & 0 \\ 0 & 0.01125 \end{pmatrix}.$$

With the gradient $(0.38, 0.18)$ of Example 11.1, the natural gradient is $(0.38/0.025,\ 0.18/0.01125) = (15.2, 16.0)$. In general, dividing the gradient components $\gamma^k \Pr(\text{reach } s)\, p_s(1-p_s)\, \Delta Q_s$ by the Fisher entries cancels both the visitation and the sensitivity:

$$F^{-1}\nabla_z J = \frac{1}{1-\gamma}\big(\Delta Q_{\text{Start}},\ \Delta Q_{\text{Middle}}\big), \qquad \Delta Q_s = Q^\pi(s, \text{go}) - Q^\pi(s, \text{exit}).$$

The natural gradient depends only on how much better going on is than exiting, not on how confident the policy already is or how often the state is visited. In particular it does not vanish on the plateaus of the Trail's logit landscape (Chapter 11): at $z = (-5, 0)$, where the plain gradient is about 0.01, the natural gradient is still $(15.2, 16)$.

Figure 12.3 compares the two methods. With step sizes chosen so that both reach $J > 3.2$ from the uniform policy in six iterations ($\eta = 8$ for the plain gradient, $\eta = 0.05$ for the natural gradient), the natural gradient needs between 9 and 13 iterations from each of the four starting points of Figure 12.3a, while the plain gradient needs 46 from $z = (-5, -5)$ and 40 from $(-6, 0)$. The plain gradient first moves only z_0, because the Middle decision matters only once Middle is reached, and crawls across two plateaus in turn; the natural gradient moves both logits at once along almost straight lines.

Invariance. The natural gradient's most important property is that it does not depend on how the policy is parameterized.

Proposition 12.5 The natural gradient is invariant to reparameterization

Let $\theta = \varphi(\vartheta)$ be a smooth reparameterization with invertible Jacobian $\Phi = \partial\theta/\partial\vartheta$ at the current point, and let g_θ, F_θ and g_ϑ, F_ϑ be the gradients and Fisher matrices in the two parameterizations. Then $\Phi F_\vartheta^{-1} g_\vartheta = F_\theta^{-1} g_\theta$: a natural gradient step in ϑ moves θ, and hence the policy, the same way to first order as a natural gradient step in θ.

Proof. By the chain rule $g_\vartheta = \Phi^\top g_\theta$ and $\nabla_\vartheta \log\pi = \Phi^\top \nabla_\theta \log\pi$, so $F_\vartheta = \Phi^\top F_\theta \Phi$. Hence $F_\vartheta^{-1} g_\vartheta = \Phi^{-1} F_\theta^{-1} \Phi^{-\top}\Phi^\top g_\theta = \Phi^{-1} F_\theta^{-1} g_\theta$, and multiplying by Φ gives the claim. ∎

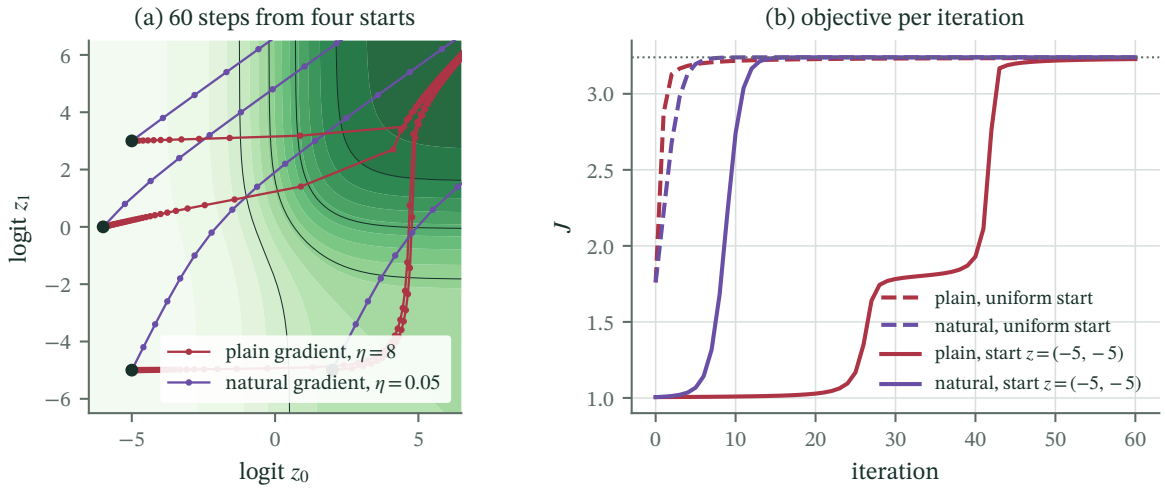


Figure 12.3. Plain and natural gradient ascent with exact gradients on the Trail, in logit space. (a) Sixty iterations from four starting points (dots), each marker one iteration; the background shows J. Plain gradient steps ($\eta = 8$) crawl along the plateaus where the policy is nearly deterministic; natural gradient steps ($\eta = 0.05$) move both logits at once. (b) The objective per iteration from the uniform policy (dashed), where both methods are equally fast, and from $z = (-5, -5)$ (solid), where the plain gradient stalls twice.

The plain gradient has no such property: $\Phi\, g_\vartheta = \Phi\Phi^\top g_\theta$, which differs from g_θ unless Φ is orthogonal. Whether the Trail's policy is written with probabilities or with logits, natural gradient ascent changes the policy in the same way (Exercise 12.7); plain gradient ascent does not.

Softmax policies and multiplicative weights. For tabular softmax policies the natural gradient has a closed form, which connects it to policy iteration.

Proposition 12.6 Natural gradient ascent for tabular softmax policies

Let $\pi_\theta(a \mid s) \propto \exp(\theta_{s,a})$ and suppose $d^{\pi_\theta}(s) > 0$ for all s. Then $u = A^{\pi_\theta}/(1-\gamma)$, the advantage table divided by $1-\gamma$, solves $F(\theta)\,u = \nabla_\theta J(\theta)$, and the natural gradient step $\theta \leftarrow \theta + \eta\, u$ gives the policy

$$\pi_{\text{new}}(a \mid s) \propto \pi_\theta(a \mid s)\, \exp\Big(\frac{\eta}{1-\gamma} A^{\pi_\theta}(s,a)\Big).$$

Every solution of $Fu = \nabla_\theta J$, in particular $F^+\nabla_\theta J$, differs from this u by a constant in each state and gives the same new policy.

Proof. The score of (s,a) is $e_a - \pi_\theta(\cdot \mid s)$ in the block of state s and zero elsewhere, so F is block diagonal with blocks $d^{\pi_\theta}(s)\big(\operatorname{diag}(\pi_s) - \pi_s\pi_s^\top\big)$, where $\pi_s = \pi_\theta(\cdot \mid s)$. By the policy gradient theorem, the gradient's block is $\frac{d^{\pi_\theta}(s)}{1-\gamma}\,\pi_s \odot A_s$, with $A_s = A^{\pi_\theta}(s,\cdot)$ and $\odot$ the elementwise product. Then

$$d^{\pi_\theta}(s)\big(\operatorname{diag}(\pi_s) - \pi_s\pi_s^\top\big)\frac{A_s}{1-\gamma} = \frac{d^{\pi_\theta}(s)}{1-\gamma}\big(\pi_s \odot A_s - \pi_s\,(\pi_s^\top A_s)\big) = \frac{d^{\pi_\theta}(s)}{1-\gamma}\,\pi_s \odot A_s,$$

because $\pi_s^\top A_s = \sum_a \pi_\theta(a \mid s) A^{\pi_\theta}(s,a) = 0$. The null space of each block is spanned by the all-ones vector, and adding a constant to all logits of a state does not change the softmax. ■

The update multiplies each action's probability by the exponential of its advantage: a *multiplicative weights* or *soft policy iteration* step. As $\eta \to \infty$ it becomes the greedy policy improvement step of policy iteration (Theorem 4.6); for finite η it is a cautious version of it. Chapter 14 proves that this update converges to an optimal policy at a rate that does not depend on the size of the state space (Theorem 14.6; see also Section 14.4 and A. Agarwal et al. (2021)). Chapter 11 showed a second route to the same direction: the weights of a compatible critic are $(1-\gamma)F^{-1}\nabla J$, which is how natural actor-critic methods estimate the natural gradient from samples (Peters and Schaal 2008a).

12.4.1 Why the Fisher matrix measures KL divergence ●●

Proposition 12.7 The KL divergence is locally quadratic, with the Fisher matrix as its Hessian

Let p_θ be a distribution on a finite set with $p_\theta(x) > 0$, twice continuously differentiable in θ. Then

$$D_{\mathrm{KL}}(p_\theta \,\|\, p_{\theta+\Delta}) = \tfrac{1}{2}\,\Delta^\top F_p(\theta)\,\Delta + o(\|\Delta\|^2),$$

where $F_p(\theta) = \mathbb{E}_{x\sim p_\theta}\big[\nabla_\theta \log p_\theta(x)\, \nabla_\theta \log p_\theta(x)^\top\big] = -\mathbb{E}_{x\sim p_\theta}\big[\nabla_\theta^2 \log p_\theta(x)\big]$.

Proof. Let $f(\Delta) = D_{\mathrm{KL}}(p_\theta \,\|\, p_{\theta+\Delta}) = \sum_x p_\theta(x)\big[\log p_\theta(x) - \log p_{\theta+\Delta}(x)\big]$. Then $f(0) = 0$. Its gradient at 0 is $-\sum_x p_\theta(x)\nabla_\theta \log p_\theta(x) = -\nabla_\theta \sum_x p_\theta(x) = 0$. Its Hessian at 0 is $-\sum_x p_\theta(x)\,\nabla_\theta^2 \log p_\theta(x)$, and since $\nabla^2 \log p = \nabla^2 p/p - \nabla \log p\, \nabla \log p^\top$ and $\sum_x \nabla_\theta^2 p_\theta(x) = \nabla_\theta^2 1 = 0$, this Hessian equals $\sum_x p_\theta(x)\, \nabla \log p_\theta(x)\, \nabla \log p_\theta(x)^\top = F_p(\theta)$. Taylor's theorem gives the expansion. ■

Applying the proposition to $\pi_\theta(\cdot \mid s)$ in each state and averaging over $s \sim d^{\pi_\theta}$ gives the approximation used above, with $F(\theta)$ of (12.3). Two remarks. The KL divergence is not symmetric, but its second-order expansions in the two directions agree, so either direction leads to the same natural gradient. And the identity $F = -\mathbb{E}[\nabla^2 \log p]$ is what makes trust-region methods practical: the Hessian of the average KL can be computed exactly from the action distributions of the sampled states, without sampling actions.

12.5 Trust region policy optimization

Trust region policy optimization (TRPO) turns the monotonic improvement theorem into a practical algorithm (Schulman et al. 2015). It keeps the theorem's structure, a surrogate objective plus a limit on how far the policy may move, but makes three changes: the penalty, whose constant is far too large, becomes a constraint with a radius δ chosen by hand; the maximum KL divergence over all states becomes an average over the sampled states; and exact advantages become

estimates. Each iteration solves

$$\max_{\theta} \frac{1}{N}\sum_t \rho_t(\theta)\hat{A}_t \quad \text{subject to} \quad \frac{1}{N}\sum_t D_{\mathrm{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t)\big) \le \delta, \tag{12.5}$$

over a batch of N transitions collected with $\pi_{\theta_{\text{old}}}$, with $\rho_t(\theta) = \pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$. Typical radii are $\delta = 0.01$ to 0.05.

An approximate solution. Linearize the objective and take the quadratic approximation of the constraint around θ_{old}. The objective's gradient is the policy gradient estimate $g = \frac{1}{N}\sum_t \hat{A}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ (Proposition 12.1), and the constraint's Hessian is the Fisher matrix F averaged over the sampled states (Proposition 12.7). The approximate problem is exactly the steepest-ascent problem of Section 12.4, so its solution is the natural gradient step (12.4), $\Delta\theta^* = \sqrt{2\delta/(g^\top F^{-1} g)}\, F^{-1} g$. TRPO is natural gradient ascent with an adaptive step size that keeps every update the same size in KL.

Conjugate gradient. For a network with d parameters, F is a $d \times d$ matrix, and for d in the millions it cannot even be stored. TRPO never forms it. The *conjugate gradient method* (Hestenes and Stiefel 1952) solves a linear system $Fx = g$ with a symmetric positive definite F using only products Fv. Starting from $x = 0$ with residual $r = g$ and search direction $p = r$, each iteration computes

$$\alpha = \frac{r^\top r}{p^\top F p}, \quad x \leftarrow x + \alpha p, \quad r_{\text{new}} = r - \alpha F p, \quad p \leftarrow r_{\text{new}} + \frac{r_{\text{new}}^\top r_{\text{new}}}{r^\top r}\, p, \quad r \leftarrow r_{\text{new}}.$$

The search directions are mutually conjugate ($p_i^\top F p_j = 0$ for $i \neq j$), so in exact arithmetic the method finishes in at most d iterations (Exercise 12.8); in practice about ten iterations give a good enough direction. A small *damping* term, solving $(F + cI)x = g$ with c around 0.01 to 0.1, keeps the system well conditioned.

Fisher-vector products. Each conjugate gradient iteration needs one product Fv. Since F is the Hessian of the average KL divergence, Fv is a Hessian-vector product, which automatic differentiation computes by differentiating $\nabla_\theta \overline{D_{\mathrm{KL}}}(\theta)^\top v$ once more (Pearlmutter 1994). For a softmax policy there is an equally cheap form: $F = \frac{1}{N}\sum_i J_i^\top \big(\mathrm{diag}(\pi_i) - \pi_i \pi_i^\top\big) J_i$, where J_i is the Jacobian of the logits of state s_i with respect to θ. The product $J_i v$ costs one forward-mode pass through the network, multiplying by the small matrix $\mathrm{diag}(\pi_i) - \pi_i \pi_i^\top$ costs nothing, and multiplying by $J_i^\top$ is one ordinary backward pass. This is how `rlcourse.ppo` computes it.

Line search. The quadratic model may be inaccurate, so TRPO backtracks: it tries $\theta_{\text{old}} + \beta^j \Delta\theta^*$ for $j = 0, 1, 2, \ldots$ with $\beta = 0.5$, and accepts the first candidate whose sampled surrogate improves and whose average KL divergence is within the radius (implementations allow a small tolerance; ours accepts up to 1.5δ).

Algorithm 12.1 TRPO (trust region policy optimization)

Input: KL radius δ, conjugate gradient iterations K, damping $c \ge 0$, backtracking factor $\beta \in (0, 1)$, discount γ, GAE parameter λ.

1. Initialize the policy parameters θ and the critic V_w.
2. For $k = 0, 1, 2, \ldots$:
   1. Collect a batch of N transitions with π_θ; set $\theta_{\text{old}} \leftarrow \theta$.
   2. Compute advantages $\hat{A}_t$ with GAE ((11.5)) and normalize them over the batch.
   3. $g \leftarrow \frac{1}{N} \sum_t \hat{A}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ at θ_{old}.
   4. Run K conjugate gradient iterations on $(F + cI)x = g$, using Fisher-vector products at θ_{old}.
   5. $\Delta \leftarrow \sqrt{2\delta / x^\top (F + cI) x}\; x$.
   6. For $j = 0, 1, 2, \ldots$: if $\theta_{\text{old}} + \beta^j \Delta$ improves the sampled surrogate and keeps the average KL within δ, set $\theta \leftarrow \theta_{\text{old}} + \beta^j \Delta$ and stop the search.
   7. Fit the critic to the targets $\hat{A}_t + V_w(s_t)$ by a few epochs of regression.

Output: the policy π_θ and the critic V_w.

In step 5, $x^\top (F + cI)x$ costs one more Fisher-vector product; if no candidate passes the test in step 6, the policy is left unchanged. On CartPole, with $\delta = 0.01$ and batches of 8×256 transitions, our compact TRPO kept the average KL per update at 0.007 on average, never above 0.010, and rarely needed to backtrack; its returns are shown in Figure 12.5.

TRPO works, and its updates are remarkably stable, but it is expensive and awkward. Each update costs about ten Fisher-vector products plus the line search's forward passes, roughly the price of dozens of gradient evaluations. It takes one step per batch, so it cannot exploit the many cheap minibatch steps that make deep learning efficient. And the second-order machinery does not mix easily with architectures that share parameters between the policy and the value function, or with auxiliary losses. Approximations of F by Kronecker factors (Martens and Grosse 2015) make natural gradient steps cheaper, as in ACKTR (Wu et al. 2017). PPO takes a different road: it drops the second-order machinery altogether.

12.6 Proximal policy optimization

Proximal policy optimization (PPO) keeps TRPO's goal, improve the surrogate without moving the policy too far, but achieves it with a first-order method (Schulman, Wolski, et al. 2017). It alternates between collecting a batch with the current policy and running several epochs of minibatch stochastic gradient ascent on a modified surrogate, the *clipped objective*:

$$L^{\text{CLIP}}(\theta) = \frac{1}{N} \sum_t \min\Big(\rho_t(\theta) \hat{A}_t,\; \text{clip}\big(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon\big) \hat{A}_t \Big), \tag{12.6}$$

where $\text{clip}(x, l, u) = \min(\max(x, l), u)$ and ϵ is the *clip range*, typically 0.2. (The clip range ϵ is unrelated to the exploration rate ε of ε-greedy policies.)

Reading the objective. Take one sample and consider its term as a function of its ratio ρ (Figure 12.4).

- *Positive advantage* ($\hat{A} > 0$): the action was better than average, and the surrogate rewards making it more likely. Up to $\rho = 1 + \epsilon$ the term is $\rho \hat{A}$, as in the plain surrogate. Beyond that point it is flat at $(1 + \epsilon)\hat{A}$: there is no further incentive to raise this action's probability.

- *Negative advantage* ($\hat{A} < 0$): the action was worse than average, and the surrogate rewards making it less likely. Down to $\rho = 1-\epsilon$ the term is $\rho\hat{A}$; below that point it is flat at $(1-\epsilon)\hat{A}$.

The minimum makes the clipping one-sided. Moving a ratio in the *unfavorable* direction (raising the probability of a bad action, or lowering that of a good one) is always penalized at the full, unclipped rate, so the gradient pulls such ratios back. Only progress in the favorable direction stops counting once it exceeds the clip range. As a result, for every sample, $\min(\rho\hat{A}, \text{clip}(\rho)\hat{A}) \leq \rho\hat{A}$: L^{CLIP} is a pessimistic version of the surrogate, equal to it while all ratios lie in $[1-\epsilon, 1+\epsilon]$. At $\theta = \theta_{\text{old}}$ all ratios are 1, so L^{CLIP} has the same value and gradient as the surrogate there (Exercise 12.10), and the first epoch's first step is an ordinary policy gradient step. The gradient of one sample's term is

$$\frac{\partial}{\partial\theta}\min\big(\rho\hat{A}, \text{clip}(\rho, 1-\epsilon, 1+\epsilon)\hat{A}\big) = \begin{cases} 0 & \text{if } \hat{A} > 0 \text{ and } \rho > 1+\epsilon, \\ 0 & \text{if } \hat{A} < 0 \text{ and } \rho < 1-\epsilon, \\ \hat{A}\,\rho\,\nabla_\theta \log \pi_\theta(a \mid s) & \text{otherwise.} \end{cases}$$

Samples whose ratios have already moved far enough in the right direction drop out of the update; the others keep pushing.

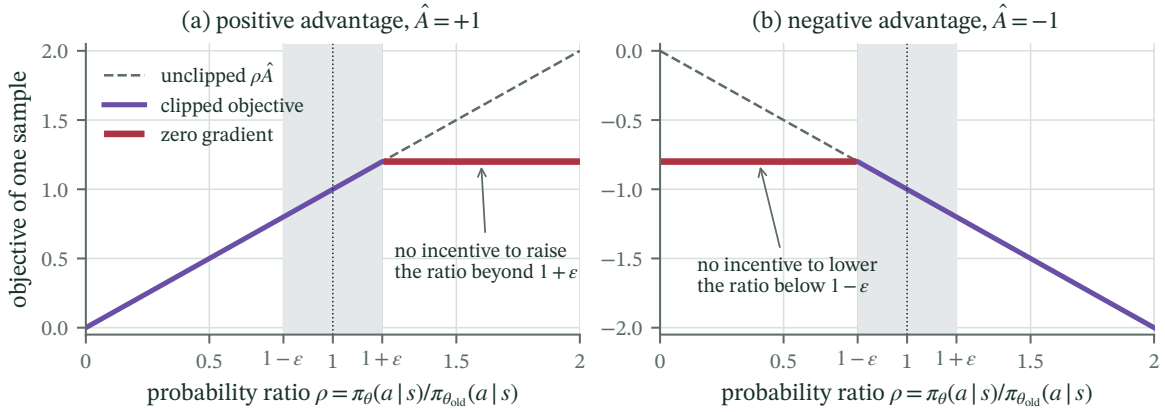


Figure 12.4. PPO's clipped objective for one sample, as a function of the probability ratio ρ, with $\epsilon = 0.2$. The dashed line is the unclipped surrogate $\rho\hat{A}$. (a) For a positive advantage the objective stops increasing at $\rho = 1+\epsilon$. (b) For a negative advantage it stops increasing (as ρ falls) at $\rho = 1-\epsilon$. On the thick segments the gradient is zero; in the unfavorable direction the objective keeps the full slope, so moving that way is never free.

Example 12.1 Three samples

Suppose $\pi_{\text{old}}(a \mid s) = 0.5$ for each of three samples, $\epsilon = 0.2$, and after a few minibatch steps the new probabilities are as follows.

Table 12.1. The clipped objective for three samples.

Sample	$\hat{A}$	$\pi_\theta(a \mid s)$	ρ	$\rho\hat{A}$	clip$(\rho)\hat{A}$	objective	gradient?
1	+2	0.65	1.3	2.6	2.4	2.4	none
2	−2	0.30	0.6	−1.2	−1.6	−1.6	none
3	−2	0.65	1.3	−2.6	−2.4	−2.6	pulls ρ back down

Sample 1 has already gained more than the clip range allows and stops contributing. Sample 2 has lost probability, as it should, beyond the clip range and also stops contributing. Sample 3 moved the wrong way, perhaps because a shared network was pushed by other samples, and its term keeps the unclipped slope, so the next steps lower its probability again.

♦

The KL-penalty variant. Schulman, Wolski, et al. (2017) also proposed a version closer to the theory, which subtracts a KL penalty from the surrogate,

$$L^{\mathrm{KL}}(\theta) = \frac{1}{N}\sum_t \Big[\rho_t(\theta)\hat{A}_t - \beta\, D_{\mathrm{KL}}\big(\pi_{\theta_{\mathrm{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t)\big)\Big],$$

and adapts β after each iteration to hit a target divergence d_{targ}: if the measured average KL is below $d_{\mathrm{targ}}/1.5$, halve β; if it is above $1.5\, d_{\mathrm{targ}}$, double it. In their continuous-control experiments, the clipped objective with $\epsilon = 0.2$ scored best among the variants compared, which included fixed and adaptive KL penalties, and clipping became the standard.

Algorithm 12.2 PPO with clipping and GAE

Input: N environments, rollout length T, epochs K, minibatch size M, clip range ϵ, discount γ, GAE parameter λ, learning rates, entropy coefficient $c_H \geq 0$.

1. Initialize the actor π_θ and the critic V_w; reset the environments.
2. For each iteration:
   1. Run π_θ for T steps in each environment; store states, actions, rewards, termination and truncation flags, $\log \pi_\theta(a_t \mid s_t)$ and $V_w(s_t)$.
   2. Compute GAE advantages $\hat{A}_t$ ((11.5); for terminations, truncations and rollout ends see Section 11.7) and critic targets $\hat{R}_t = \hat{A}_t + V_w(s_t)$.
   3. Set $\theta_{\mathrm{old}} \leftarrow \theta$.
   4. For K epochs, for each minibatch B of M transitions from a random partition of the NT transitions:
      1. Normalize the advantages over B.
      2. $\rho_t \leftarrow \exp\big(\log \pi_\theta(a_t \mid s_t) - \log \pi_{\theta_{\mathrm{old}}}(a_t \mid s_t)\big)$ for $t \in B$.
      3. Take a gradient step that increases $\frac{1}{M}\sum_{t\in B}\big[\min(\rho_t\hat{A}_t, \mathrm{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\hat{A}_t) + c_H\, \mathcal{H}(\pi_\theta(\cdot \mid s_t))\big]$.
      4. Take a gradient step that decreases $\frac{1}{2M}\sum_{t\in B}\big(V_w(s_t) - \hat{R}_t\big)^2$.

5. Optionally stop the epochs early if the approximate KL divergence exceeds a target.

Output: the actor π_θ and the critic V_w.

The log-probabilities stored in step 2.1 define $\pi_{\theta_{\text{old}}}$; they are not recomputed. The advantages and targets of step 2.2 are fixed for the whole iteration. With a shared network for actor and critic, steps 2.4.3 and 2.4.4 become one step on the combined objective $L^{\text{CLIP}} - c_V L^V + c_H \mathcal{H}$, with a value coefficient c_V. The settings of the original paper for continuous control were $T = 2048$ steps per iteration, $K = 10$ epochs, minibatches of 64, Adam with learning rate 3×10^{-4}, $\gamma = 0.99$, $\lambda = 0.95$ and $\epsilon = 0.2$ (Schulman, Wolski, et al. 2017). Our CartPole runs use 8 environments $\times$ 128 steps, 10 epochs, minibatches of 256 and a learning rate of 10^{-3}.

What PPO buys is simplicity: a few lines added to an actor-critic implementation, ordinary first-order optimizers, minibatches, any architecture. One batch supports $K \times NT/M$ gradient steps (here $10 \times 1024/256 = 40$), instead of TRPO's single step, and the clipping keeps those steps from wandering far, most of the time. PPO has been used at very large scale, for example to train agents for the game Dota 2 (Berner et al. 2019), and it is the standard reinforcement-learning step of RLHF for language models (Section 22.2).

LAB

Lab 12.1: Natural gradients, trust regions and PPO (`labs/ch12_ppo.py`, about 60 minutes). Run `python3 labs/ch12_ppo.py`. You will compute KL divergences and the Fisher matrix of the Trail's policy, compare the surrogate with the true objective and check the improvement bound, run plain and natural gradient ascent from a plateau, evaluate the clipped objective by hand, train PPO on CartPole while watching its diagnostics, and train PPO with a Gaussian policy on Pendulum. The "Going further" ideas include ablating PPO's implementation details and running the compact TRPO.

TRY IT INTERACTIVELY

Module 7 of the interactive course (`Apps_claude/RL Course/index.html`) shows one unlucky REINFORCE episode at $\eta = 10$, the KL divergence between two Trail policies, a big step that fools the surrogate, and the clipped objective for both signs of the advantage.

12.7 PPO in practice

PPO's paper describes an objective; PPO's success comes from an implementation. Engstrom et al. (2020) found that much of PPO's reported advantage over TRPO on continuous-control benchmarks came from *code-level optimizations* in the reference implementation rather than from the clipped objective itself. Huang, Dossa, Raffin, et al. (2022) catalogued 37 implementation details of the standard implementation, and Andrychowicz et al. (2021) measured the effect of dozens of design choices in a large-scale study. The details that matter most are these.

Advantage normalization. Standardize the advantages within each minibatch (zero mean, unit

variance). This makes the size of the policy update independent of the scale of the rewards, and of how that scale changes during training.

Value function loss and clipping. The critic is trained on the GAE targets $\hat{R}_t$. Some implementations also clip the critic's change, using $\max\big((V_w - \hat{R})^2, (V_{\text{old}} + \text{clip}(V_w - V_{\text{old}}, -\epsilon_V, \epsilon_V) - \hat{R})^2\big)$ by analogy with the policy. Evidence that this helps is mixed (Engstrom et al. 2020; Andrychowicz et al. 2021).

Observation and reward normalization. Inputs are standardized with running means and variances. Rewards are divided by a running estimate of the standard deviation of the discounted return, which keeps the critic's targets of order one; our Pendulum runs use this.

Learning-rate annealing. The learning rate decreases linearly to zero over training. Besides fine-tuning, this shrinks the policy change per update late in training, as the KL curve of Figure 12.5 shows.

Orthogonal initialization. Weight matrices are initialized as scaled orthogonal matrices, with a small gain (such as 0.01) for the policy's output layer, so that the initial policy is close to uniform (or, for Gaussian policies, has a mean close to zero), and gain 1 for the value output.

Gradient clipping and Adam settings. The global norm of each minibatch gradient is clipped (to 0.5 in the reference implementation and in ours), and Adam's ϵ is set to 10^{-5} instead of the usual 10^{-8}.

Early stopping. If the approximate KL divergence between the old and the current policy exceeds a target (such as 0.01 to 0.05), the remaining epochs of the iteration are skipped. This is the most direct way of enforcing a trust region in PPO.

Diagnostics. Four quantities, logged after every iteration, reveal most problems.

- The *approximate KL divergence* $\frac{1}{N}\sum_t\big[(\rho_t - 1) - \log \rho_t\big]$. Each term is non-negative, since $\log x \le x - 1$, and since $\mathbb{E}_{a\sim\pi_{\text{old}}}[\rho] = 1$, its expectation is exactly $D_{\text{KL}}(\pi_{\text{old}} \,\|\, \pi_\theta)$; Section 22.7 compares it with other estimators. Values far above the intended trust region signal steps that are too large.
- The *clip fraction*, the share of samples with $|\rho_t - 1| > \epsilon$ after the update. A few percent is typical; a large fraction means the epochs push most ratios out of the clip range.
- The *explained variance* of the critic, $1 - \text{Var}(\hat{R} - V)/\text{Var}(\hat{R})$, computed before the update: 1 is a perfect critic, 0 is no better than a constant, negative is worse. A critic that never learns makes all advantages noise.
- The *entropy* of the policy, whose collapse to zero long before the returns stop improving warns of premature convergence.

Figure 12.5 shows PPO and TRPO on CartPole with the book's NumPy networks. PPO averages returns of 288 after 25,000 steps and 442 after 50,000, and all four seeds reach the maximum of 500 by 75,000 steps. TRPO keeps pace early (288 and 423) but dips later in training. PPO's average KL per update is 0.004 (never above 0.017), smaller than TRPO's 0.007, and its clip fraction is about 4% on average, highest in the first updates. The critic's explained variance rises from about 0 to 0.6 over the first ten updates; late in training, when every episode reaches the time limit, the targets vary little and the statistic becomes noisy.

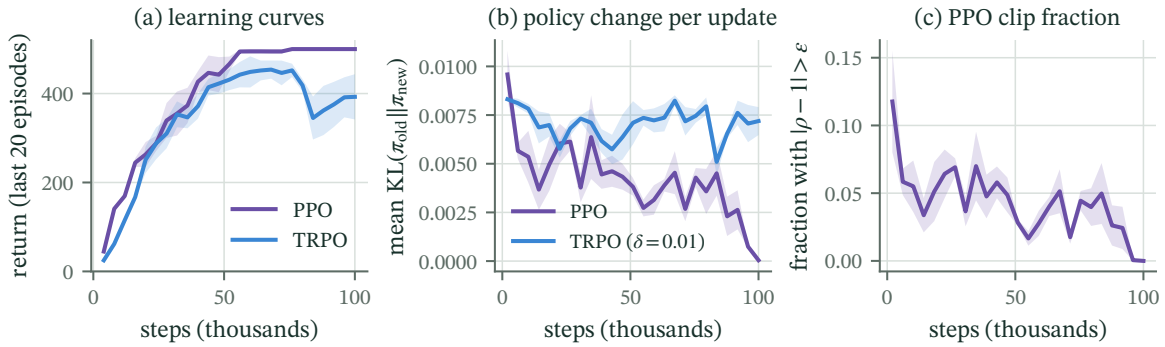


Figure 12.5. PPO and TRPO on CartPole (four seeds; bands show ±1 standard error). (a) Average return of the last 20 episodes: PPO reaches the maximum of 500 in all seeds by 75,000 steps. (b) Average KL divergence between the policies before and after each update: TRPO holds it near its radius $\delta = 0.01$, while PPO's falls toward zero as its learning rate is annealed. (c) PPO's clip fraction, the share of samples whose ratio left $[1-\epsilon, 1+\epsilon]$ during the update.

For continuous actions, PPO uses the Gaussian policy of Chapter 11 with a state-independent $\log\sigma$. Figure 12.6 shows it on Pendulum, where the agent must swing a pendulum up and hold it upright with a bounded torque; rewards are negative costs, and the first episodes of our runs score about −1,200. With $\gamma = 0.9$, reward scaling, 4 environments × 512 steps per iteration and σ initialized at 0.61, the average return improves to about −310 after 40,000 steps and −176 after 100,000, while σ shrinks to about 0.29. Acting with the mean action instead of sampling, the three trained policies score −146, −130 and −146 over ten evaluation episodes.

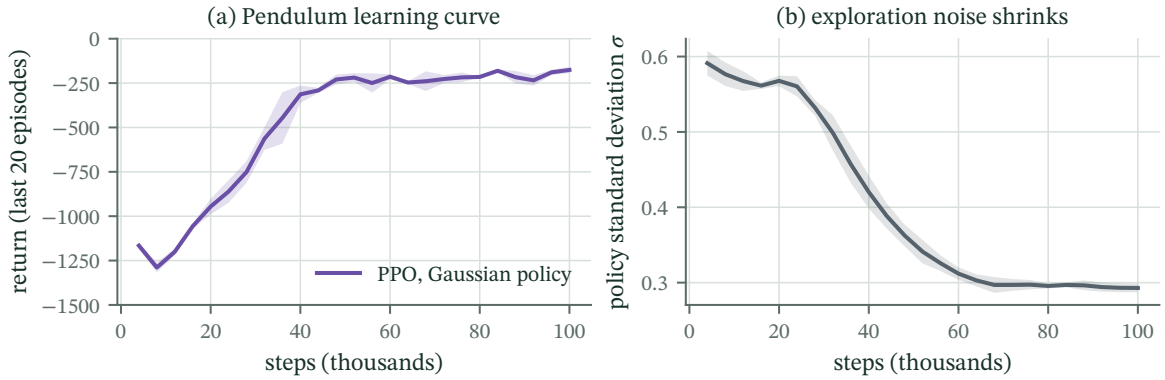


Figure 12.6. PPO with a Gaussian policy on Pendulum (three seeds; bands show ±1 standard error). (a) Average return of the last 20 episodes. (b) The policy's standard deviation σ, a learned parameter, shrinks as the policy becomes confident; the annealed learning rate freezes it near 0.29.

PITFALL

PPO's diagnostics are easy to misread. A clip fraction of zero does not mean that all is well: it can mean that the learning rate is so small that nothing changes. With the estimator of the diagnostics above, a negative approximate KL is a bug (each of its terms is non-negative), usually a sign error in the log-ratio. And a rising return with an explained variance near zero means that the policy is learning from advantages that are mostly noise, which tends to stall later.

12.8 What PPO really optimizes ●●●

Clipping is often described as keeping the new policy within a trust region of the old one. It does not, at least not directly. The clipped objective removes the incentive to move a sampled ratio beyond the clip range, but it does not stop the ratio from getting there. Shared parameters move all ratios together, minibatch steps overshoot, and samples with small advantages are pulled along by others. In our CartPole runs, at least one sample's ratio exceeded $1 + \epsilon = 1.2$ after 74% of the updates; the median of the largest ratio per update was 1.25, and the largest seen was 1.98. Exercise 12.13 constructs a two-state example in which a ratio reaches 1.54 while the clipped objective has stopped changing. On continuous-control benchmarks, Engstrom et al. (2020) found ratios well outside the clip range and KL divergences that clipping alone did not control, and Ilyas et al. (2020) found that the gradient estimates, value baselines and surrogate landscapes of deep policy gradient methods often differ substantially from what the derivations assume.

What then keeps PPO's updates small? Mostly the small learning rate, the modest number of epochs and minibatch steps, annealing and early stopping, with clipping as one ingredient among several. Clipping does matter: removing it from our CartPole configuration, with all else unchanged, raised the average KL per update from 0.004 to 0.28 and slowed learning (Exercise 12.15). But its effect is empirical, not a guarantee.

PPO as approximate mirror descent. A cleaner way to see what TRPO and PPO approximate is to solve the penalized surrogate exactly in a tabular setting. For each state, maximizing the old policy's advantage under the new policy minus a KL penalty,

$$\max_{\pi(\cdot|s)} \sum_a \pi(a \mid s) A^{\pi_{\text{old}}}(s,a) - \beta\, D_{\text{KL}}\big(\pi(\cdot \mid s) \,\|\, \pi_{\text{old}}(\cdot \mid s)\big),$$

has the closed-form solution $\pi_{\text{new}}(a \mid s) \propto \pi_{\text{old}}(a \mid s) \exp\big(A^{\pi_{\text{old}}}(s,a)/\beta\big)$ (Exercise 12.12): a mirror-descent step with the KL divergence as the distance, identical to the natural gradient step of Proposition 12.6 with $\eta/(1-\gamma) = 1/\beta$. TRPO and PPO can be read as approximating this ideal update in parameter space, with a limited number of gradient steps. Tomar et al. (2022) make the connection explicit in mirror descent policy optimization, and Kuba et al. (2022) develop a general framework of such updates with convergence guarantees; the theory of policy mirror descent is in Chapter 14.

Variants and current practice. PPO remains the default on-policy algorithm in deep reinforcement learning, especially where simulation is cheap and massively parallel. Variants change one ingredient at a time: phasic policy gradient separates policy and value training into alternating

phases (Cobbe, Hilton, et al. 2021); for language models, PPO is combined with a KL penalty toward a reference model (Ouyang et al. 2022), and critic-free variants such as GRPO keep the clipped ratio but replace the learned critic by group-relative baselines (Shao et al. 2024), as described in Algorithm 22.2. Asymmetric and adaptive clip ranges, token-level versus sequence-level ratios, and the choice of KL estimator have become active topics in that setting; Chapter 22 discusses them.

RESEARCH NOTE

Open questions: What trust region, if any, does PPO enforce, and can its behavior be characterized as an optimization method rather than as a heuristic? How should the clip range, the number of epochs and the KL penalty be set as a function of batch size and network size? How stale can data be, in asynchronous large-scale training, before the surrogate's first-order guarantee stops being useful? And which parts of the theory of Section 12.3 survive with estimated advantages and function approximation?

12.9 Summary

- Plain policy gradient steps are measured in parameter space, where the same step can change the policy a lot or hardly at all; with noisy estimates, large steps push policies into near-deterministic regions from which the vanishing gradient never rescues them. On the Trail with $\eta = 10$, most REINFORCE runs end stuck.
- The surrogate objective L predicts a new policy's return from the old policy's data, using the old policy's state distribution and importance ratios. It matches J to first order; on the Trail the error is exactly a product of the policy changes in the two states.
- (●●) The monotonic improvement theorem bounds $J - L$ by the policy change, in total variation or KL, times a constant proportional to $1/(1-\gamma)^2$. Maximizing the resulting lower bound improves monotonically, but the constant is far too large for practice.
- The natural gradient $F^{-1}\nabla J$ is steepest ascent when steps are measured by KL divergence. It is invariant to reparameterization, does not stall on plateaus, and for tabular softmax policies equals a multiplicative-weights update on advantages. On the Trail it is $(\Delta Q_{\text{Start}}, \Delta Q_{\text{Middle}})/(1-\gamma)$.
- TRPO maximizes the surrogate subject to an average-KL constraint: a natural gradient step computed by conjugate gradient with Fisher-vector products, followed by a line search. It is stable but expensive.
- PPO replaces the constraint by a clipped objective that removes the incentive to move ratios beyond $[1-\epsilon, 1+\epsilon]$ in the favorable direction, and optimizes it with several epochs of minibatch steps. Its implementation details (advantage normalization, learning-rate annealing, initialization, gradient clipping, early stopping) matter as much as the objective, and its diagnostics (approximate KL, clip fraction, explained variance, entropy) should always be logged.
- (●●●) Clipping does not bound the ratios or the KL; PPO's updates stay small mostly because of its optimization settings. TRPO and PPO approximate a KL-regularized mirror-descent step.

Chapter 13 turns to off-policy actor-critic methods for continuous control, which reuse old data through a learned action-value critic instead of importance ratios.

12.10 Further Reading

The natural gradient was introduced for learning in general by Amari (1998) and for policies by Kakade (2001), whose short paper also shows the connection to policy iteration. Kakade and Langford (2002) analyze conservative policy iteration, the ancestor of every trust-region method; Schulman et al. (2015) derive TRPO from their analysis, and Achiam et al. (2017) give the tighter, average-divergence form of the bound together with its use for constrained problems. Peters and Schaal (2008a) develop natural actor-critic methods, and Martens and Grosse (2015) and Wu et al. (2017) show how to make natural gradients affordable for large networks. The conjugate gradient method is due to Hestenes and Stiefel (1952), and the trick for computing Hessian-vector products to Pearlmutter (1994).

PPO is presented in Schulman, Wolski, et al. (2017), a short and readable paper; the GAE paper (Schulman et al. 2016) shows TRPO with value functions on locomotion tasks. Before implementing PPO, read Huang, Dossa, Raffin, et al. (2022), which lists the details of the reference implementation one by one, and Engstrom et al. (2020) and Andrychowicz et al. (2021), which measure which of them matter.

The theory of natural policy gradient and policy mirror descent, including global convergence rates, is in A. Agarwal et al. (2021) and Chapter 14.

(●●●) Ilyas et al. (2020) examine how well the quantities that deep policy gradient methods estimate match their idealized counterparts. Tomar et al. (2022) and Kuba et al. (2022) develop the mirror-descent view of trust-region methods, and Cobbe, Hilton, et al. (2021) is an example of a successful PPO variant. For PPO and its descendants in language models, start with Ouyang et al. (2022) and Shao et al. (2024), and continue in Chapter 22.

12.11 Exercises

Exercise 12.1 ● KL divergences between Trail policies

(a) Compute $D_{\mathrm{KL}}(\mathrm{Bern}(0.5)\,\|\,\mathrm{Bern}(0.7))$ and $D_{\mathrm{KL}}(\mathrm{Bern}(0.7)\,\|\,\mathrm{Bern}(0.5))$, and compare both with the quadratic approximation $\frac{1}{2}p(1-p)\,\Delta z^2$ at $p = 0.5$, where Δz is the change of logit. (b) Compute the KL divergence of the unit logit step $z \to z+1$ from $z = 0$ and from $z = 4$, and the ratio of the two. (c) Explain in two sentences why a fixed step size in logit space is either reckless or slow.

Exercise 12.2 ● The surrogate by hand

The old Trail policy is uniform and the new one has $p_0 = 0.7$, $p_1 = 0.4$ ($\gamma = 0.9$). (a) Compute J of the new policy and the surrogate L. (b) Check the formula $J - L = \gamma\,\Delta Q_{\mathrm{Middle}}(p_0 - q_0)(p_1 - q_1)$. (c) Is the surrogate optimistic or pessimistic here, and why?

Exercise 12.3 ●● First-order agreement and second-order disagreement

(a) Prove Proposition 12.1 directly for the Trail from the formula for L, by differentiating with respect to the logits at $z = z_{\text{old}}$. (b) Compute the Hessian of $J - L$ with respect to (p_0, p_1) and show that it is not zero. Which entry is non-zero, and what does it represent?

Exercise 12.4 ●● The Fisher matrix of a softmax

(a) For a softmax distribution $\pi = \text{softmax}(h)$ over n actions, show that the Fisher matrix with respect to the logits is $\text{diag}(\pi) - \pi\pi^\top$. (b) Show that it is singular, identify its null space, and explain why the singularity is harmless for natural gradient steps. (c) Show that, for any vector A with $\pi^\top A = 0$, $u = A$ solves $(\text{diag}(\pi) - \pi\pi^\top)\, u = \pi \odot A$, the identity at the heart of Proposition 12.6.

Exercise 12.5 ● A natural gradient by hand

For the Trail with $p_0 = 0.8$, $p_1 = 0.5$ and $\gamma = 0.9$: (a) compute $d^\pi(\text{Start})$, $d^\pi(\text{Middle})$ and the Fisher matrix; (b) compute the natural gradient from the gradient $\nabla_z J = (0.2432, 0.288)$ (computed in the exercises of Chapter 11), and compare it with the natural gradient at the uniform policy; (c) compute the step (12.4) for $\delta = 0.01$.

Exercise 12.6 ●● How loose is the bound?

At the uniform Trail policy with $\gamma = 0.9$: (a) compute ϵ_A and the constant $C = 4\gamma\epsilon_A/(1-\gamma)^2$ of Theorem 12.3; (b) for the new policy $p_0 = p_1 = 0.6$, compute J, L, the maximum KL divergence and both lower bounds; (c) the Trail's episodes last at most three steps. Sketch how the proof of Lemma 12.2 could be sharpened for problems whose episodes last at most H steps, and what replaces the factor $1/(1-\gamma)$.

Exercise 12.7 ●● Invariance on the Trail

Parameterize the Trail's policy directly by the probabilities $\vartheta = (p_0, p_1)$ instead of the logits. (a) Compute the Fisher matrix F_p and the gradient $\nabla_p J$. (b) Show that $F_p^{-1}\nabla_p J = \big(p_0(1-p_0)\Delta Q_{\text{Start}},\ p_1(1-p_1)\Delta Q_{\text{Middle}}\big)/(1-\gamma)$, and check Proposition 12.5 against the logit natural gradient. (c) At $p_0 = 0.8$, $p_1 = 0.5$, show that the plain gradients in the two parameterizations do *not* correspond: the change of (p_0, p_1) induced by a small plain gradient step in the logits points in a different direction from $\nabla_p J$. Why do they agree at the uniform policy?

Exercise 12.8 ●● Conjugate gradient

(a) Run two iterations of the conjugate gradient method by hand on $Ax = b$ with $A = \begin{pmatrix} 4 & 1 \\ 1 & 3 \end{pmatrix}$ and $b = (1, 2)$, starting from $x = 0$, and check that the result is the exact solution. (b) Prove that the residuals of successive iterations are orthogonal, $r_1^\top r_0 = 0$, and explain why this

implies termination in at most n iterations for an $n \times n$ system. (c) In TRPO, each conjugate gradient iteration costs one Fisher-vector product. Estimate the cost of one TRPO update with ten iterations and a line search of three trials, in units of one gradient evaluation on the batch.

Exercise 12.9 • The clipped objective case by case

With $\epsilon = 0.2$, compute the clipped objective of one sample and its derivative with respect to ρ for $(\rho, \hat{A}) = (0.7, 1), (1.1, 1), (1.5, 1), (0.7, -1), (1.1, -1), (1.5, -1)$. For each case, say in words whether the update will continue to push the ratio and in which direction.

Exercise 12.10 •• Clipping is pessimistic

(a) Show that $\min\big(\rho\hat{A}, \mathrm{clip}(\rho, 1-\epsilon, 1+\epsilon)\hat{A}\big) \le \rho\hat{A}$ for all $\rho > 0$ and $\hat{A}$, with equality whenever $\rho \in [1-\epsilon, 1+\epsilon]$. (b) Conclude that L^{CLIP} and the sampled surrogate have the same value and gradient at $\theta = \theta_{\mathrm{old}}$. (c) Is L^{CLIP} (plus a constant) a lower bound on the true objective J? Explain what it does and does not bound.

Exercise 12.11 • Reading PPO's logs

Diagnose each situation and propose a fix. (a) In the first updates, the approximate KL is 0.15 and the clip fraction 0.6. (b) After 50 updates the entropy has fallen to 0.02 nats while the returns stopped improving at a mediocre level. (c) The explained variance stays below 0 and the value loss grows. (d) The approximate KL is sometimes negative. (e) The clip fraction is exactly 0 in every update and the returns do not change.

Exercise 12.12 •• The KL-penalized update and its temperature

(a) For one state with finitely many actions, show that the maximizer of $\sum_a \pi(a)A(a) - \beta\, D_{\mathrm{KL}}(\pi \,\|\, \pi_{\mathrm{old}})$ over distributions π is $\pi(a) \propto \pi_{\mathrm{old}}(a)\, e^{A(a)/\beta}$. (b) For two actions with π_{old} uniform and $A = (+1, -1)$, compute the new probability of the first action, its KL divergence from π_{old} and the gain $\sum_a \pi(a)A(a)$ for $\beta \in \{0.5, 1, 2, 4\}$. (c) Explain how the adaptive rule for β in Section 12.6 behaves if the measured KL stays above $1.5\, d_{\mathrm{targ}}$, and why it settles when the KL lies in $[d_{\mathrm{targ}}/1.5,\ 1.5\, d_{\mathrm{targ}}]$.

Exercise 12.13 •• Clipping does not bound the ratios

Two states A and B have one-dimensional features $x_{\mathrm{A}} = 1$ and $x_{\mathrm{B}} = 3$, and a two-action policy uses the shared logit θx for action R, so $\pi_\theta(\mathrm{R} \mid s) = \sigma(\theta x_s)$. The batch has two samples, both with action R: one in A with $\hat{A} = +1$ and one in B with $\hat{A} = 0$. Start from $\theta_{\mathrm{old}} = 0$ and $\epsilon = 0.2$. (a) Show that gradient ascent on L^{CLIP} stops changing θ once $\rho_{\mathrm{A}} = 1.2$, and find that θ. (b) Compute ρ_{B} there and the KL divergences between the old and new policies in both states. (c) What does this say about clipping as a trust region, and which diagnostic would reveal it?

Exercise 12.14 ●● The Fisher matrix of a Gaussian policy

For a one-dimensional Gaussian policy $\mathcal{N}(\mu, \sigma^2)$ with parameters $(\mu, \log\sigma)$, compute the Fisher matrix from the scores $(a-\mu)/\sigma^2$ and $(a-\mu)^2/\sigma^2 - 1$ of Chapter 11, and the natural gradient of a function with ordinary gradient $(g_\mu, g_{\log\sigma})$. Interpret the result: how does the natural gradient scale the update of the mean when the policy is very confident (small σ)?

Exercise 12.15 ● Which details matter?

Programming. In Lab 12.1, train PPO on CartPole for 100,000 steps with three seeds in each of these configurations: the full configuration; without advantage normalization; without orthogonal initialization; without gradient clipping; without learning-rate annealing; with value clipping; without the clip (plain surrogate, same epochs); and with one epoch instead of ten. Report the final return, the average return over training and the average KL per update. Which details matter on CartPole, and which would you expect to matter more on harder problems?

Exercise 12.16 ●●● Trust regions in RLHF

In reinforcement learning from human feedback, PPO's per-update trust region (clipping relative to π_{old}) coexists with a KL penalty toward a fixed reference model ((22.1)). Explain the different roles of the two divergences: which one is part of the *objective* and which one only shapes the *optimization*? Read Ouyang et al. (2022) and Shao et al. (2024) and describe how each handles the two. Propose an experiment, on a toy problem such as the bandit-like setting of Section 22.1, that separates their effects.

Exercise 12.17 ●●● Is PPO a trust-region method?

Read Engstrom et al. (2020) and Ilyas et al. (2020). Summarize the evidence that PPO's performance depends on code-level optimizations and that clipping does not keep the policy within a trust region. Then design an experiment with Lab 12.1 that measures, over training, the largest ratio, the average KL and the KL in the worst state of each batch, with and without early stopping on the KL. What result would convince you that clipping is, or is not, the reason PPO is stable?

CHAPTER 13

Off-Policy Actor-Critic for Continuous Control

WHAT YOU WILL LEARN

- Explain why value-based methods struggle when actions are continuous, and how an actor that learns to maximize a critic removes the difficulty.
- State the deterministic policy gradient theorem, compute it by hand on small problems, and prove it (●●).
- Implement DDPG and TD3 with a replay buffer, target networks and the critic's action gradient, and explain what each of TD3's three changes fixes.
- Define the maximum-entropy objective, soft value functions and the soft Bellman equation, and derive the Boltzmann form of the soft-optimal policy.
- Prove that soft policy iteration improves monotonically and converges at rate γ (●●).
- Implement soft actor-critic with a tanh-squashed Gaussian policy, its log-probability correction and automatic temperature tuning, and train it on Pendulum.
- Choose between on-policy and off-policy methods for a given problem, and describe recent off-policy developments and their open questions (●●●).

Many control problems have continuous actions: the torques at a robot's joints, the steering angle of a car, the thrust of a drone's rotors, the dose of a drug. Chapter 11 and Chapter 12 handled continuous actions with Gaussian policies and on-policy methods such as PPO. Those methods learn from fresh experience of the current policy, use each batch for a few gradient steps, and then throw it away. When experience is cheap, that is a reasonable price for stability. When it is expensive, as on a real robot or in a slow simulator, it is wasteful. Deep Q-learning (Chapter 10) is frugal with experience, because it learns from a replay buffer of old transitions, but it needs $\max_a Q(s, a)$ at every step. With ten discrete actions the maximum is a lookup. With a continuous action it is an optimization problem.

This chapter combines the two families. An off-policy *critic* is trained from replay, as in DQN, and an *actor* learns to output the action that the critic rates highest. The actor improves by following the gradient of the critic with respect to the action, a quantity given by the *deterministic policy gradient*. Three algorithms built on this idea are the workhorses of continuous control: DDPG, its corrected successor TD3, and soft actor-critic (SAC). Along the way we meet two ideas that reach beyond continuous control. The first is that a critic which is maximized by its own actor tends to overestimate, and that this bias can be controlled. The second is *maximum-entropy reinforcement learning*, which changes the objective so that the agent stays as random as it can

afford to be.

Section 13.1 explains the difficulty and the actor-critic answer, and Section 13.2 states the deterministic policy gradient theorem (its proof, in Section 13.2.1, is ●●). Section 13.3 and Section 13.4 build DDPG and TD3. Section 13.5 introduces maximum-entropy RL, with a ●● subsection on soft policy iteration, and Section 13.6 turns it into SAC. Section 13.7 runs all three methods on Pendulum with the book's NumPy library, and Section 13.8 compares on-policy and off-policy methods. Section 13.9 (●●, with ●●● subsections) surveys later developments. On a first reading, take Section 13.1 to 13.8 and skip the proofs.

13.1 Continuous actions

Our running example is *Pendulum*, a classic control task (`rlcourse.envs.Pendulum`, with the dynamics of Gymnasium's `Pendulum-v1`). A pole hangs from a motorized pivot. The agent observes the angle ψ of the pole from upright (as $\cos\psi$ and $\sin\psi$) and its angular velocity $\dot{\psi}$, and chooses a torque $u \in [-2, 2]$. Each step costs $\psi^2 + 0.1\,\dot{\psi}^2 + 0.001\,u^2$, so the reward is the negative of that cost and is never positive. The best place to be is upright and still. The motor is too weak to lift the pole directly from the bottom, so a good policy first swings it back and forth to build up energy, then catches it at the top and balances it. Episodes start at a random angle and are cut off after 200 steps; the task itself has no terminal states. A policy that picks torques at random earns about -1250 per episode, and a good one about -150 (it cannot avoid the cost of swinging up from a random start).

13.1.1 The trouble with the maximum

Q-learning and DQN need the maximum over actions twice: to act greedily, $a = \arg\max_a Q(s, a)$, and to compute every learning target, $r + \gamma \max_{a'} Q(s', a')$. With a continuous action, each maximum is an optimization problem that must be solved at every step and for every transition in every minibatch. There are three classic ways around this, and each has a cost.

Discretize the actions. Pendulum's torque could be split into 21 levels, and DQN would work. But the number of combinations grows exponentially with the number of action dimensions: a robot arm with 7 joints and 10 levels per joint has 10^7 joint actions, one output unit each. *Restrict the form of the critic.* If $Q(s, a)$ is a quadratic function of a whose coefficients depend on s, its maximizer has a closed form; this is the normalized advantage function method of Gu et al. (2016). The price is a critic that cannot represent values with several peaks in a. *Optimize numerically.* A sampling method such as the cross-entropy method can approximately maximize any critic, as in the QT-Opt robot-grasping system of Kalashnikov et al. (2018). The price is many critic evaluations per decision.

13.1.2 The actor-critic answer

The approach of this chapter is to *learn* the maximization. Alongside the critic $Q_w(s, a)$ we train a second network, the *actor* $\mu_\theta(s)$, whose output is an action. We want $\mu_\theta(s)$ to be the action that maximizes $Q_w(s, \cdot)$, so we adjust θ to increase $Q_w(s, \mu_\theta(s))$, by gradient ascent through the

critic. The left panel of Figure 13.1 shows the idea for one state: the critic's slope at the actor's current action says which way to move the action, and the chain rule turns "move the action" into "change the actor's parameters". The critic, in turn, learns the values of the actor's behavior from replayed transitions, with the target $r + \gamma Q_w(s', \mu_\theta(s'))$ in place of $r + \gamma \max_{a'} Q(s', a')$. The actor is an *amortized* optimizer: instead of solving a new maximization at every step, it learns a function that solves all of them at once, approximately.

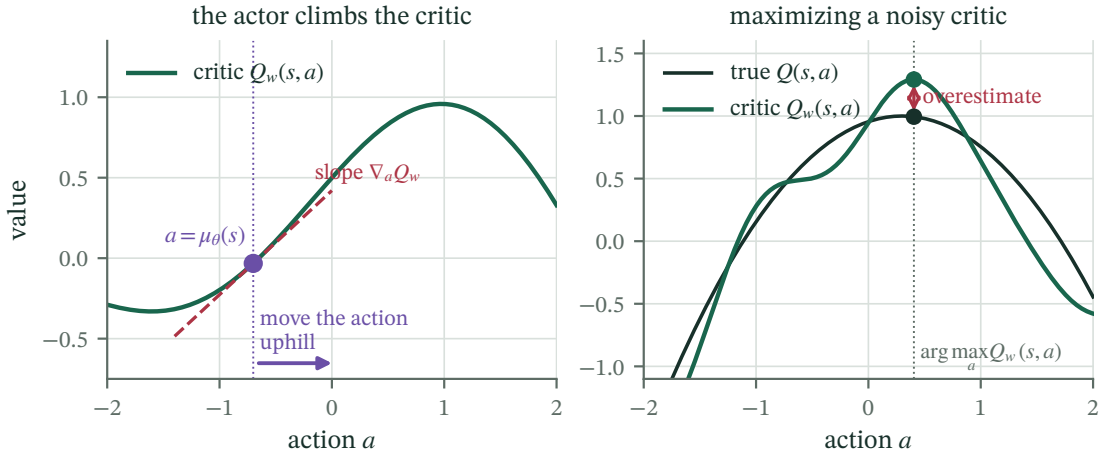


Figure 13.1. Left: the actor climbs the critic. For a fixed state, the actor's current action $a = \mu_\theta(s)$ sits on the critic's curve; the slope $\nabla_a Q_w$ there says that a larger action is better, so the actor moves its output uphill. Right: the danger of climbing a critic with errors. The critic (green) equals the true values (black) plus a smooth random error with standard deviation 0.3 at every action. At the action that maximizes the critic, the critic reads 1.29 while the true value is 0.99. Averaged over 20,000 random error functions, the critic overestimates the value of its own maximizer by 0.37, although it is unbiased at every fixed action.

The right panel of Figure 13.1 shows the danger that runs through this chapter. A critic is an estimate, with errors. An actor that climbs it is drawn toward the places where the errors are positive, so the critic's value at the actor's action is too high on average, even if the critic is unbiased everywhere else. Section 13.4 measures this effect and corrects it.

KEY IDEA

An off-policy actor-critic method learns a critic $Q_w(s, a)$ from replayed experience and an actor that outputs, in each state, the action the critic rates highest. The actor improves by following the gradient $\nabla_a Q_w(s, a)$ at its own action. This replaces the impossible $\max_a$ of Q-learning by a learned function.

13.2 The deterministic policy gradient

The policy gradient theorem of Chapter 11 (Theorem 11.1) applies to stochastic policies. It improves a policy by trying actions and making those that turned out well more probable, so it must average over the actions the policy might take. A *deterministic policy* $\mu_\theta : \mathcal{S} \to \mathcal{A}$ takes

no random actions, and its gradient takes a different, simpler form: instead of trying actions, it asks the critic which way is uphill.

Fix a deterministic policy μ_θ with parameters $\theta \in \mathbb{R}^d$ and actions in $\mathcal{A} \subseteq \mathbb{R}^m$. Its action-value function $Q^{\mu_\theta}(s,a)$ is the expected return of taking action a in state s and following μ_θ afterward, and its state value is $V^{\mu_\theta}(s) = Q^{\mu_\theta}(s, \mu_\theta(s))$. The objective is the expected return from the initial distribution, $J(\theta) = \mathbb{E}_{s_0 \sim d_0}[V^{\mu_\theta}(s_0)]$, and d^{μ_θ} is the discounted state-visitation distribution of Definition 3.8, $d^{\mu_\theta}(s) = (1-\gamma)\sum_t \gamma^t \Pr(s_t = s)$, read as a density when states are continuous. We write $\nabla_\theta \mu_\theta(s)$ for the $d \times m$ matrix whose entry (i,j) is $\partial \mu_{\theta,j}(s)/\partial \theta_i$.

Theorem 13.1 Deterministic policy gradient (Silver et al., 2014)

Assume $\gamma < 1$ and that the transition density $P(s' \mid s,a)$, the reward $r(s,a)$, the initial density d_0 and the policy $\mu_\theta(s)$ are continuous, that the derivatives $\nabla_a P(s' \mid s,a)$, $\nabla_a r(s,a)$ and $\nabla_\theta \mu_\theta(s)$ exist and are continuous, and that the densities, the reward and these derivatives are bounded. Then

$$\nabla_\theta J(\theta) = \frac{1}{1-\gamma}\, \mathbb{E}_{s \sim d^{\mu_\theta}} \Big[\nabla_\theta \mu_\theta(s)\, \nabla_a Q^{\mu_\theta}(s,a)\big|_{a=\mu_\theta(s)} \Big]. \tag{13.1}$$

In words: to improve a deterministic policy, nudge its action in every state in the direction in which the action value increases, and weight the states by how often the policy visits them, with discounting. The factor $\nabla_a Q$ is the critic's advice ("a slightly larger torque would be better here") and $\nabla_\theta \mu_\theta$ translates that advice into a change of parameters. The conditions are the regularity conditions of Silver et al. (2014); they ensure that derivatives may be taken inside the integrals over states. Compared with the stochastic policy gradient, there is no expectation over actions, which is why the deterministic gradient can be estimated with less variance; Silver et al. (2014) also show that it is the limit of the stochastic policy gradient as the variance of the policy shrinks to zero.

Example 13.1 The deterministic policy gradient by hand

A one-step problem. The state is $s = -1$ or $s = +1$ with equal probability, the agent chooses a real number a, receives the reward $r(s,a) = -(a - 2s)^2$, and the episode ends. The best action is $a = 2s$. Take the linear policy $\mu_\theta(s) = \theta s$ with $\theta = 0.5$. Because the episode ends after one step, $Q(s,a) = r(s,a)$, and

$$\nabla_a Q(s,a)\big|_{a=\mu_\theta(s)} = -2(\theta s - 2s) = 3s, \qquad \nabla_\theta \mu_\theta(s) = s.$$

In state $s = +1$ the critic says "increase the action" (slope $+3$); in state $s = -1$ it says "decrease it" (slope -3). Both pieces of advice mean "increase θ", because $\nabla_\theta \mu_\theta(-1) = -1$ flips the sign. The deterministic policy gradient is the average, $\frac{1}{2}(3 \cdot 1 + (-3)(-1)) = 3$. (Only time 0 contributes: the terminal state that follows has no action to improve, so (13.1) reduces to an average over the initial distribution.) The check: $J(\theta) = \mathbb{E}[-(\theta s - 2s)^2] = -(\theta - 2)^2$, whose derivative at $\theta = 0.5$ is $-2(0.5 - 2) = 3$. Lab 13.1 repeats the check on a linear-quadratic problem that unfolds over time, where the theorem's discounted visitation matters (Exercise 13.9).

♦

Training the actor off-policy. In practice the actor is trained on states from a replay buffer, collected by older and noisier versions of the policy, and with a learned critic Q_w in place of Q^{μ_θ}. For a minibatch of states $s_1, \dots, s_B$ the actor's update direction is

$$\hat{g} = \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta \mu_\theta(s_i)\, \nabla_a Q_w(s_i, a)\big|_{a=\mu_\theta(s_i)}. \tag{13.2}$$

This is a gradient of the off-policy objective $\mathbb{E}_{s\sim\mathcal{D}}[Q_w(s, \mu_\theta(s))]$, the critic's average rating of the actor over the buffer's states. Two approximations hide in it. The states come from the buffer rather than from d^{μ_θ}, and the critic replaces the true action values. For the first, Silver et al. (2014) drop the term that accounts for the effect of θ on the action values themselves, appealing to the argument that Degris et al. (2012) gave for stochastic off-policy actor-critic, where the approximate gradient still improves the policy, at least in the tabular case. Note what is *not* needed: no importance-sampling ratio over actions appears, because the gradient involves no expectation over actions. The buffer's actions are used only by the critic, which learns $Q_w(s, a)$ for whatever action was taken, as Q-learning does. Neither approximation is corrected in DDPG, TD3 or SAC, and they work best when the buffer's states are not too different from those the current policy visits.

13.2.1 Proof of the deterministic policy gradient theorem ●●

The proof follows the Bellman-equation proof of the stochastic policy gradient theorem. Write μ for μ_θ. Under the regularity conditions all value functions are bounded and differentiable, and derivatives may be exchanged with the integrals below.

Proof of Theorem 13.1. The state value satisfies $V^\mu(s) = Q^\mu(s, \mu_\theta(s)) = r(s, \mu_\theta(s)) + \gamma \int P(s' \mid s, \mu_\theta(s))\, V^\mu(s')\, ds'$. The parameter θ enters in three places: in the action taken in s (inside r and inside P) and in the future behavior (inside V^μ). Differentiating with the chain rule,

$$\begin{aligned}
\nabla_\theta V^\mu(s) &= \nabla_\theta \mu_\theta(s) \Big[\nabla_a r(s, a) + \gamma \int \nabla_a P(s' \mid s, a)\, V^\mu(s')\, ds' \Big]_{a=\mu_\theta(s)} \\
&\quad + \gamma \int P(s' \mid s, \mu_\theta(s))\, \nabla_\theta V^\mu(s')\, ds' \\
&= \underbrace{\nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s, a)\big|_{a=\mu_\theta(s)}}_{=:\, h(s)} + \gamma \int P(s' \mid s, \mu_\theta(s))\, \nabla_\theta V^\mu(s')\, ds',
\end{aligned}$$

because $Q^\mu(s, a) = r(s, a) + \gamma \int P(s' \mid s, a) V^\mu(s')\, ds'$, so the bracket is $\nabla_a Q^\mu(s, a)$ with the policy held fixed. The function $g = \nabla_\theta V^\mu$ therefore satisfies a Bellman equation, $g = h + \gamma P^\mu g$, whose "reward" is h and whose dynamics are those of μ. The map $g \mapsto h + \gamma P^\mu g$ is a γ-contraction on bounded functions (as in Theorem 4.3) and h is bounded, so its unique fixed point is the series obtained by unrolling it:

$$\nabla_\theta V^\mu(s) = \sum_{t=0}^{\infty} \gamma^t\, \mathbb{E}_\mu\big[h(s_t) \mid s_0 = s\big].$$

Averaging over $s_0 \sim d_0$ gives $\nabla_\theta J(\theta) = \sum_t \gamma^t \mathbb{E}_\mu[h(s_t)] = \frac{1}{1-\gamma} \mathbb{E}_{s\sim d^\mu}[h(s)]$, by the definition of d^μ. This is (13.1). ■

13.3 Deep deterministic policy gradient (DDPG)

The deterministic policy gradient was known in 2014, but making it work with neural networks needed the stabilizers of DQN. Lillicrap et al. (2016) combined them into *deep deterministic policy gradient* (DDPG) and reported that one set of hyperparameters learned policies for more than twenty simulated physics tasks. DDPG has four ingredients.

Replay. Every transition (s, a, r, s', d) goes into a buffer $\mathcal{D}$, and both networks learn from uniformly sampled minibatches, which breaks the correlation between consecutive transitions and reuses each one many times (Section 10.2.1). The flag d is 1 only if s' is a true terminal state.

A critic with a target. The critic minimizes the squared error to the bootstrapped target

$$y = r + \gamma(1 - d)\, Q_{\bar{w}}\big(s', \mu_{\bar{\theta}}(s')\big), \tag{13.3}$$

computed with *target networks* $Q_{\bar{w}}$ and $\mu_{\bar{\theta}}$: slowly moving copies of the critic and the actor (Section 10.2.2). Compared with DQN's target, the maximum over a' is replaced by the target actor's choice.

Soft target updates. Instead of copying the networks every few thousand steps, DDPG moves the targets a little after every update, $\bar{w} \leftarrow (1-\tau)\bar{w} + \tau w$ and $\bar{\theta} \leftarrow (1-\tau)\bar{\theta} + \tau\theta$, with a small averaging rate such as $\tau = 0.005$. This is *Polyak averaging*: the target is an exponentially weighted average of past networks whose average age is $(1 - \tau)/\tau = 199$ updates (Exercise 13.3).

Exploration noise. A deterministic actor never tries anything new by itself, so DDPG acts with $a = \mathrm{clip}(\mu_\theta(s) + \sigma\xi, -1, 1)$, adding Gaussian noise $\xi \sim \mathcal{N}(0, I)$ of scale σ (we keep the actor's outputs in $[-1, 1]$ and scale them to the environment's torque range when acting). The original DDPG used temporally correlated noise from an Ornstein–Uhlenbeck process (Uhlenbeck and Ornstein 1930); Fujimoto et al. (2018) found no benefit from it over independent Gaussian noise, which is now the usual choice.

Algorithm 13.1 DDPG

Input: discount γ, averaging rate τ, step sizes η_Q and η_μ, exploration noise σ, batch size B.

1. Initialize the actor μ_θ and the critic Q_w; set $\bar{\theta} \leftarrow \theta$, $\bar{w} \leftarrow w$; empty the buffer $\mathcal{D}$.
2. For each environment step:
   1. In state s, act with $a = \mathrm{clip}(\mu_\theta(s) + \sigma\xi, -1, 1)$, $\xi \sim \mathcal{N}(0, I)$.
   2. Observe r, s' and d ($d = 1$ only if s' is terminal, not at a time limit); add (s, a, r, s', d) to $\mathcal{D}$.
   3. Sample a minibatch $\{(s_i, a_i, r_i, s'_i, d_i)\}_{i=1}^{B}$ from $\mathcal{D}$.
   4. Targets: $y_i \leftarrow r_i + \gamma(1 - d_i)\, Q_{\bar{w}}(s'_i, \mu_{\bar{\theta}}(s'_i))$.
   5. Critic: $w \leftarrow w - \eta_Q \nabla_w \frac{1}{2B} \sum_i \big(Q_w(s_i, a_i) - y_i\big)^2$.
   6. Actor: $\theta \leftarrow \theta + \eta_\mu \frac{1}{B} \sum_i \nabla_\theta \mu_\theta(s_i)\, \nabla_a Q_w(s_i, a)\big|_{a=\mu_\theta(s_i)}$.
   7. Targets: $\bar{w} \leftarrow (1-\tau)\bar{w} + \tau w$ and $\bar{\theta} \leftarrow (1-\tau)\bar{\theta} + \tau\theta$.
   8. $s \leftarrow s'$, or reset the environment if the episode ended (terminated or truncated).

Output: the actor μ_θ, used without noise.

In practice the gradient steps use Adam rather than plain gradient descent, the first thousand or so actions are uniformly random to fill the buffer, and updates begin only when the buffer holds enough transitions. Step 6 is the only unusual computation. It needs $\nabla_a Q_w$, the gradient of the critic with respect to one of its *inputs*. The small network library of the book returns exactly this from `backward()` (see Section C.2), so the actor's update takes a few lines. In the snippet below, `s` is a batch of states and the critic's input is the concatenation $[s, a]$.

```
a = actor.forward(s)                        # mu_theta(s) for the batch, shape (B, m)
q = critic.forward(np.hstack([s, a]))       # Q_w(s, mu_theta(s)), shape (B, 1)
dx = critic.backward(np.ones_like(q))       # dQ/d(input) per row: [dQ/ds, dQ/da]
critic.zero_grad()                          # discard the critic's own parameter gradients
dq_da = dx[:, obs_dim:]                     # keep the action part
actor.zero_grad()
actor.backward(-dq_da / len(s))             # chain rule for the loss -mean Q
actor_opt.step(actor.grads)
```

The last two lines are the chain rule of (13.2): backpropagating $-\nabla_a Q_w$ through the actor produces $-\nabla_\theta \mu_\theta(s)\, \nabla_a Q_w$, the gradient of the loss $-Q_w(s, \mu_\theta(s))$, averaged over the batch.

Why DDPG is brittle. DDPG can learn quickly, but its results are known to vary widely across random seeds, hyperparameters and implementation details (Henderson et al. 2018). Three weaknesses reinforce each other. The actor is trained only through the critic, so any error in the critic's slope becomes an error in the policy. The actor climbs the critic, so it seeks out the critic's overestimates (Figure 13.1, right). And a deterministic actor can lock onto a narrow peak of the critic that does not exist in the true values. In our Pendulum runs (Section 13.7) DDPG learns as fast as SAC, but several of its seeds lose a good policy for a while after having found it.

TRUNCATION IS NOT TERMINATION

Pendulum never terminates: episodes are cut after 200 steps only so that the agent sees many starting positions. At the cut, the next state still has a value, so the stored flag must be $d = 0$ and the target must bootstrap from $Q_{\bar{w}}(s', \mu_{\bar{\theta}}(s'))$. Storing $d = 1$ at the time limit, a common bug, asks the critic to predict a return that depends on how many steps remain, which the state does not reveal: the same state gets targets that mean different things. Exercise 13.8 computes how far the learned values then fall from the truth, and Section 6.8 discusses the bug in general.

CHECKPOINT

1. Why does DDPG need exploration noise, while an agent with a stochastic policy does not need any added?
2. What do the averaging rates $\tau = 1$ and $\tau = 0$ mean?
3. In the actor update, why must `critic.zero_grad()` be called after the critic's backward pass?

Answers.

1. A deterministic actor always takes the same action in a state, so without noise the critic never sees what other actions are worth. A stochastic policy explores through its own randomness.
2. $\tau = 1$ copies the online networks into the targets after every update (no target networks at all); $\tau = 0$ freezes the targets forever.
3. The backward pass also accumulates the gradient of $Q_w(s, \mu_\theta(s))$ with respect to the critic's parameters. That gradient belongs to no loss of the critic, and must not leak into the critic's next update.

13.4 Twin delayed DDPG (TD3)

Section 6.7 and Section 10.4 showed that the maximum of noisy estimates is biased upward: even if every estimate is unbiased, the largest one is likely to be one whose error happens to be positive. An actor-critic method takes no explicit maximum, but its actor does gradient ascent on the critic and so performs the maximization implicitly. The model in the right panel of Figure 13.1 makes the effect concrete. The true values are $Q(s, a) = 1 - \frac{1}{2}(a - 0.3)^2$, and the critic adds a smooth random error with standard deviation 0.3 at every action, so the critic is unbiased at each fixed action. Yet at the action that maximizes the critic, the critic's value exceeds the true value by 0.37 on average (over 20,000 random error functions), and acting greedily on the critic loses 0.10 of true value. Thrun and Schwartz (1993) described this mechanism for Q-learning with function approximation.

Bootstrapping makes it worse. The target $r + \gamma Q_{\bar{w}}(s', \mu_{\bar{\theta}}(s'))$ evaluates the critic exactly at the action the actor has chosen, so the overestimate enters every target and is then learned, propagated backward to earlier states, and maximized again. Fujimoto et al. (2018) measured this overestimation in DDPG on simulated locomotion tasks and proposed *twin delayed DDPG* (TD3), which changes DDPG in three ways.

1. Clipped double Q-learning. TD3 trains two critics, Q_{w_1} and Q_{w_2}, with different random initializations, and both regress on the same target, which uses the smaller of the two target critics:

$$y = r + \gamma(1 - d) \min_{j=1,2} Q_{\bar{w}_j}(s', \tilde{a}'). \tag{13.4}$$

If the two critics' errors were independent, the minimum would remove the upward bias: in the model of Figure 13.1, taking the minimum of two independent critics at the maximizer of the first gives an average error of -0.03 instead of $+0.37$. The price is a slight *under*estimation, and it is a good trade. An overestimated action attracts the actor, which then feeds the error back into the targets; an underestimated action is avoided by the actor, so its error is not amplified. Why not use Double DQN's remedy, letting one network choose the action and another evaluate it? Because in actor-critic methods the policy changes slowly, the online and target critics are too similar for their errors to be decorrelated (Fujimoto et al. 2018).

2. Target policy smoothing. The target action is the target actor's choice plus a little clipped noise,

$$\tilde{a}' = \mathrm{clip}\big(\mu_{\bar{\theta}}(s') + \mathrm{clip}(\tilde{\sigma}\xi, -c, c),\ -1,\ 1\big), \qquad \xi \sim \mathcal{N}(0, I),$$

with, for example, $\tilde{\sigma} = 0.2$ and $c = 0.5$. The target thereby averages the critic over a small neighborhood of the chosen action. Similar actions should have similar values, and a narrow spike in the critic, which a deterministic actor could otherwise exploit, is averaged away (Exercise 13.5).

3. Delayed policy updates. The actor and the target networks are updated only once every k critic updates, typically $k = 2$. The critic then has time to adjust to the current actor before the actor exploits it again, and the targets move more slowly.

Algorithm 13.2 TD3

Input: as for DDPG, plus target noise $\tilde{\sigma}$, noise clip c and policy delay k.

1. Initialize the actor μ_θ and two critics Q_{w_1}, Q_{w_2}; set $\bar{\theta} \leftarrow \theta$, $\bar{w}_j \leftarrow w_j$; empty $\mathcal{D}$.
2. For each environment step $t = 1, 2, \ldots$:
   1. Act with $a = \text{clip}(\mu_\theta(s) + \sigma\xi, -1, 1)$; store (s, a, r, s', d) in $\mathcal{D}$.
   2. Sample a minibatch from $\mathcal{D}$.
   3. For each transition: $\tilde{a}'_i \leftarrow \text{clip}\big(\mu_{\bar{\theta}}(s'_i) + \text{clip}(\tilde{\sigma}\xi_i, -c, c), -1, 1\big)$.
   4. Targets: $y_i \leftarrow r_i + \gamma(1 - d_i)\min_{j=1,2} Q_{\bar{w}_j}(s'_i, \tilde{a}'_i)$.
   5. Update both critics by a gradient step on $\frac{1}{2B}\sum_i (Q_{w_j}(s_i, a_i) - y_i)^2$, $j = 1, 2$.
   6. If t is a multiple of k:
      1. Actor: gradient step on $\frac{1}{B}\sum_i Q_{w_1}(s_i, \mu_\theta(s_i))$, as in DDPG.
      2. Targets: $\bar{w}_j \leftarrow (1 - \tau)\bar{w}_j + \tau w_j$ and $\bar{\theta} \leftarrow (1 - \tau)\bar{\theta} + \tau\theta$.

Output: the actor μ_θ, used without noise.

The actor follows the first critic only; the minimum enters only the targets. Figure 13.2 shows what clipped double Q-learning does on Pendulum. Every 2,000 steps we compare the critic's estimate $Q_w(s, \mu_\theta(s))$, averaged over 100 states from the replay buffer, with the true discounted value of the current policy from those states, computed by simulating the policy. Early in training both critics are hundreds too high, for a reason that has nothing to do with maximization: they start near zero, and every Pendulum reward is negative, so their initial estimates are optimistic until learning catches up. Once that has washed out, the two methods differ in the way the analysis predicts. Over the last 10,000 of 24,000 steps, DDPG's critic overestimates by 12.7 on average, while TD3's underestimates by 4.4, with values of the policies around -20 to -60.

The three changes trade some speed for stability, and how good the trade is depends on the task. On Pendulum, an easy task, removing the policy delay makes TD3 *faster*; removing clipping or smoothing left one of three seeds in a poor state at the end of training, but full TD3 also dipped once, and three seeds are too few to tell (Exercise 13.14). On harder locomotion tasks, Fujimoto et al. (2018) report that each change contributes.

13.5 Maximum-entropy reinforcement learning

Every method so far needs a separate mechanism for exploration: ε-greedy actions, an entropy bonus added to a loss, or noise added to a deterministic actor. Maximum-entropy RL takes a different route. It changes the objective so that the agent is *rewarded* for acting randomly, in

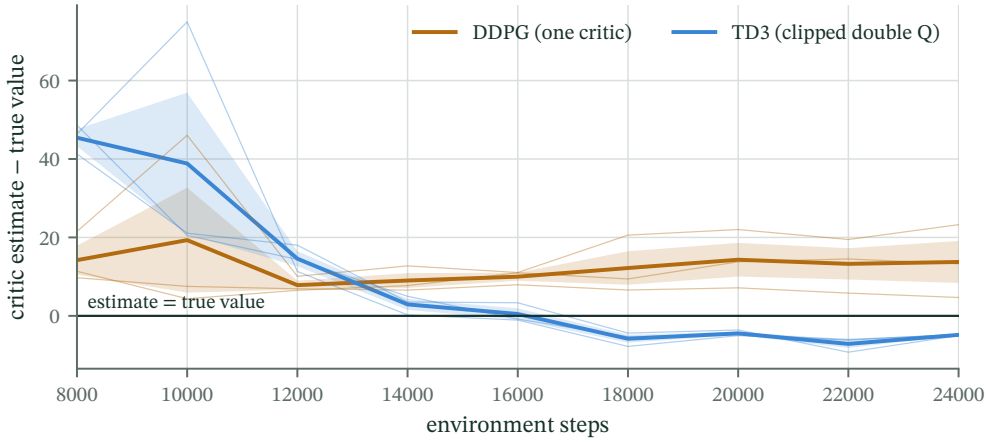


Figure 13.2. Critic estimate minus true value for DDPG and TD3 on Pendulum, from 8,000 to 24,000 steps (three seeds each; thin lines are seeds, the band is ±1 standard error of the mean). The estimate is $Q_w(s, \mu_\theta(s))$ averaged over 100 replay-buffer states; the true value is the discounted return of the current deterministic policy from the same states. DDPG's critic settles above the truth; TD3's clipped double-Q target settles slightly below it.

proportion to how little randomness costs. An agent that knows two equally good ways to reach a goal then keeps both, instead of committing to one at random; an agent facing a choice that matters little stays undecided. Besides better exploration, such policies tend to be more robust to perturbations of the environment (Eysenbach and Levine 2022), and the extra term makes the optimization landscape better behaved, a fact that Chapter 14 turns into fast convergence rates (Theorem 14.7).

The *entropy* of the policy in state s is $\mathcal{H}(\pi(\cdot \mid s)) = -\sum_a \pi(a \mid s) \log \pi(a \mid s)$ (an integral for continuous actions); it is zero for a deterministic choice among finitely many actions and $\log |\mathcal{A}|$ for the uniform one (Section A.6). The *maximum-entropy objective* adds the entropy at every visited state to the reward:

$$J_\alpha(\pi) = \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t \big(r(s_t, a_t) + \alpha\, \mathcal{H}(\pi(\cdot \mid s_t))\big)\Big], \tag{13.5}$$

where the *temperature* $\alpha > 0$ sets the exchange rate between reward and randomness. With $\alpha \to 0$ we recover the ordinary objective. The idea goes back to maximum-entropy inverse RL (Ziebart et al. 2008) and the principle of maximum causal entropy (Ziebart 2010), and it was brought to deep RL by soft Q-learning (Haarnoja et al. 2017; see also Fox et al. 2016).

Definition 13.2 Soft value functions

The *soft state value* of a policy π is the objective (13.5) started from s: $V_\alpha^\pi(s) = \mathbb{E}_\pi\big[\sum_t \gamma^t \big(r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot \mid s_t))\big) \mid s_0 = s\big]$. The *soft action value* is $Q_\alpha^\pi(s, a) = r(s, a) + \gamma\, \mathbb{E}_{s' \sim P(\cdot \mid s, a)}\big[V_\alpha^\pi(s')\big]$.

The soft action value counts the entropy of every future state but not of the current one, because the current action has already been chosen. The two are related by $V_\alpha^\pi(s) = \mathbb{E}_{a \sim \pi(\cdot \mid s)}\big[Q_\alpha^\pi(s, a) -$

$\alpha \log \pi(a \mid s)\big]$, since the expectation of $-\log \pi(a \mid s)$ is the entropy in s. Substituting this relation into the definition of Q_α^π gives the *soft Bellman equation*:

$$Q_\alpha^\pi(s,a) = r(s,a) + \gamma\, \mathbb{E}_{s' \sim P(\cdot|s,a)}\, \mathbb{E}_{a' \sim \pi(\cdot|s')}\big[Q_\alpha^\pi(s',a') - \alpha \log \pi(a' \mid s')\big]. \tag{13.6}$$

In words: the soft value of an action is its reward plus the discounted soft value of the next state, and a state's soft value is the average action value plus α times the entropy of the policy there. It is the Bellman equation of (3.8) with $\alpha \log \pi(a' \mid s')$ subtracted from each next action's value; on average this adds α times the entropy of the next state's policy.

13.5.1 The soft-optimal policy

Which policy maximizes the soft objective? In a single state with action values $q(a)$, the agent chooses a distribution p over actions to maximize $\sum_a p(a)q(a) + \alpha\mathcal{H}(p)$: high-value actions on one side, spreading out on the other. The answer is the Boltzmann (softmax) distribution.

Proposition 13.3 The soft maximum

Let $\mathcal{A}$ be finite, $q : \mathcal{A} \to \mathbb{R}$ and $\alpha > 0$. Then

$$\max_p \Big\{ \sum_a p(a)q(a) + \alpha\mathcal{H}(p) \Big\} = \alpha \log \sum_a e^{q(a)/\alpha},$$

where the maximum is over probability distributions on $\mathcal{A}$, and it is attained only at $p^*(a) = e^{q(a)/\alpha} \big/ \sum_b e^{q(b)/\alpha}$.

Proof. Let $Z = \sum_b e^{q(b)/\alpha}$, so that $q(a) = \alpha \log p^*(a) + \alpha \log Z$. For any distribution p,

$$\sum_a p(a)q(a) - \alpha \sum_a p(a) \log p(a) = \alpha \log Z - \alpha \sum_a p(a) \log \frac{p(a)}{p^*(a)} = \alpha \log Z - \alpha D_{\mathrm{KL}}(p \,\|\, p^*).$$

The KL divergence is nonnegative and zero only when $p = p^*$. ∎

The quantity $\alpha \log \sum_a e^{q(a)/\alpha}$ is a *soft maximum*: it lies between $\max_a q(a)$ and $\max_a q(a) + \alpha \log |\mathcal{A}|$ and tends to the maximum as $\alpha \to 0$. Applying Proposition 13.3 in every state gives the optimal soft values and the soft-optimal policy. The optimal soft action value Q_α^* satisfies the *soft Bellman optimality equation*

$$Q_\alpha^*(s,a) = r(s,a) + \gamma\, \mathbb{E}_{s'}\Big[\alpha \log \sum_{a'} e^{Q_\alpha^*(s',a')/\alpha}\Big], \qquad \pi_\alpha^*(a \mid s) = \frac{e^{Q_\alpha^*(s,a)/\alpha}}{\sum_b e^{Q_\alpha^*(s,b)/\alpha}}. \tag{13.7}$$

The soft-optimal policy is stochastic: it gives every action a probability that grows exponentially with the action's value, and the temperature sets how sharply. Section 13.5.2 proves that these equations have a unique solution and that the Boltzmann policy of Q_α^* maximizes (13.5).

Example 13.2 The soft Trail

Take the Trail with $\gamma = 0.9$ and temperature $\alpha = 1$. In Goal the only action is finish, so there is no entropy to earn and $V_\alpha(\text{Goal}) = 4$. In Middle the soft action values are $Q_\alpha(\text{Middle}, \text{go}) = 0.9 \cdot 4 = 3.6$ and $Q_\alpha(\text{Middle}, \text{exit}) = 2$. By Proposition 13.3,

$$V_\alpha(\text{Middle}) = \log(e^{3.6} + e^{2}) = 3.6 + \log(1 + e^{-1.6}) = 3.7839,$$
$$\pi^*_\alpha(\text{go} \mid \text{Middle}) = \frac{1}{1 + e^{-1.6}} = 0.832.$$

The soft value exceeds the best action's value, 3.6, by 0.18: the worth of keeping the option to exit. In Start, $Q_\alpha(\text{Start}, \text{go}) = 0.9 \cdot 3.7839 = 3.4055$ and $Q_\alpha(\text{Start}, \text{exit}) = 1$, so $V_\alpha(\text{Start}) = \log(e^{3.4055} + e) = 3.4919$ and $\pi^*_\alpha(\text{go} \mid \text{Start}) = 1/(1 + e^{-2.4055}) = 0.917$.

The soft-optimal policy is not optimal for the ordinary objective. Its expected return, counting rewards only, is 2.833, below the 3.24 of going all the way: the agent pays 0.41 of reward for its randomness. Figure 13.3 shows how the trade moves with the temperature. With $\alpha = 0.5$ the policy goes on with probabilities 0.989 and 0.961 and its expected return is 3.16; with $\alpha = 2$ the probabilities are 0.81 and 0.69 and the return is 2.45. As $\alpha \to 0$ the soft policy becomes the optimal deterministic one.

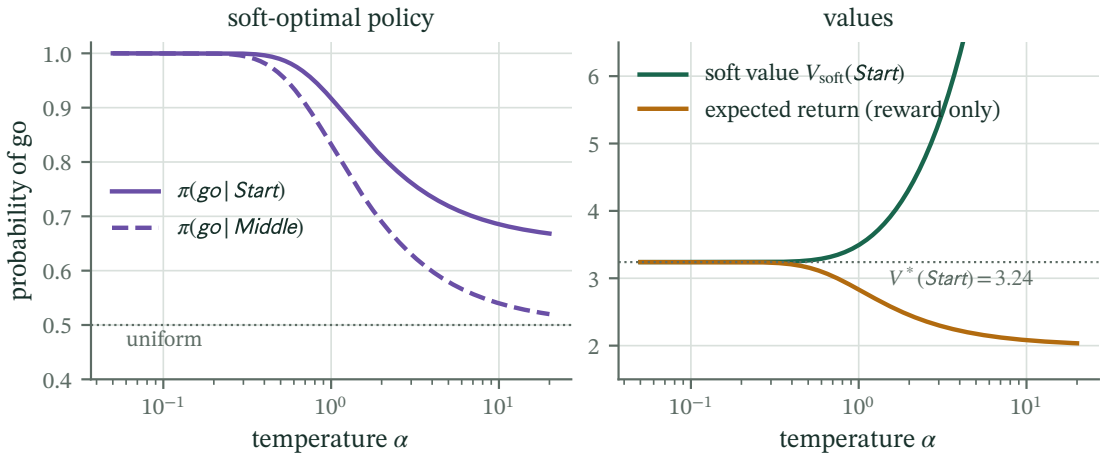


Figure 13.3. The soft Trail ($\gamma = 0.9$) as a function of the temperature, computed by soft value iteration. Left: the soft-optimal probability of go in Start and in Middle. The decision in Middle, where the stakes (3.6 against 2) are smaller relative to the temperature, becomes random first. Right: the soft value of Start, which counts entropy and grows with α, and the ordinary expected return of the soft-optimal policy, which falls from 3.24 toward the return of a coin-flipping policy.

With continuous actions the sums become integrals and $\mathcal{H}$ is the *differential* entropy, which can be negative: a one-dimensional Gaussian with standard deviation σ has entropy $\frac{1}{2}\log(2\pi e\sigma^2)$, which is below zero when $\sigma < 0.242$. The Boltzmann density $\propto e^{Q(s,a)/\alpha}$ exists whenever $\int e^{Q(s,a)/\alpha}\, da$ is finite, which holds, for example, when the actions are bounded and Q is bounded.

TRY IT INTERACTIVELY

Module 8 of the interactive course (`Apps_claude/RL Course/index.html`) lets you set the two action values of a Trail state and the temperature, and watch the soft-optimal policy and the soft value respond; its later lessons walk through SAC's target and DDPG and TD3 on the Trail.

13.5.2 Soft policy iteration ●●

Soft policy iteration alternates two steps, as policy iteration does: evaluate the current policy's soft values, then replace the policy by the Boltzmann policy of those values. We prove that it converges for finite MDPs, at the same geometric rate as ordinary policy iteration. Throughout, $\mathcal{A}$ is finite and rewards are bounded, so all soft values are bounded (entropies lie between 0 and $\log|\mathcal{A}|$).

Lemma 13.4 Soft policy evaluation

For a policy π, define the soft Bellman operator by the right-hand side of (13.6): $(\mathcal{T}_\alpha^\pi Q)(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\mathbb{E}_{a'\sim\pi}\big[Q(s',a') - \alpha \log \pi(a' \mid s')\big]$. It is monotone ($Q \le Q'$ implies $\mathcal{T}_\alpha^\pi Q \le \mathcal{T}_\alpha^\pi Q'$) and a γ-contraction in the sup norm, and its unique fixed point is Q_α^π.

Proof. $\mathcal{T}_\alpha^\pi$ is the ordinary Bellman expectation operator of π for the modified reward $r(s,a) + \gamma\alpha\mathbb{E}_{s'}[\mathcal{H}(\pi(\cdot \mid s'))]$, which is bounded. Monotonicity and the contraction property follow exactly as in Theorem 4.3, and Q_α^π is a fixed point by (13.6). ■

Lemma 13.5 Soft policy improvement

Let $\pi_{\text{new}}(\cdot \mid s)$ be the Boltzmann policy of $Q_\alpha^{\pi_{\text{old}}}(s,\cdot)$ in every state. Then $Q_\alpha^{\pi_{\text{new}}} \ge \mathcal{T}_\alpha^{\pi_{\text{new}}} Q_\alpha^{\pi_{\text{old}}} \ge Q_\alpha^{\pi_{\text{old}}}$ everywhere.

Proof. Write Q^{old} and V^{old} for the soft values of π_{old}. In every state, by Proposition 13.3 applied to $q = Q^{\text{old}}(s,\cdot)$, the new policy maximizes $\mathbb{E}_{a\sim p}[Q^{\text{old}}(s,a) - \alpha \log p(a)]$ over distributions p, so

$$\mathbb{E}_{a\sim\pi_{\text{new}}}\big[Q^{\text{old}}(s,a) - \alpha\log\pi_{\text{new}}(a \mid s)\big] \ge \mathbb{E}_{a\sim\pi_{\text{old}}}\big[Q^{\text{old}}(s,a) - \alpha\log\pi_{\text{old}}(a \mid s)\big] = V^{\text{old}}(s).$$

Hence $Q^{\text{old}}(s,a) = r(s,a) + \gamma\mathbb{E}_{s'}[V^{\text{old}}(s')] \le (\mathcal{T}_\alpha^{\pi_{\text{new}}} Q^{\text{old}})(s,a)$. Applying the monotone operator $\mathcal{T}_\alpha^{\pi_{\text{new}}}$ repeatedly to $Q^{\text{old}} \le \mathcal{T}_\alpha^{\pi_{\text{new}}} Q^{\text{old}}$ gives an increasing sequence, $Q^{\text{old}} \le \mathcal{T}_\alpha^{\pi_{\text{new}}} Q^{\text{old}} \le (\mathcal{T}_\alpha^{\pi_{\text{new}}})^2 Q^{\text{old}} \le \cdots$, which converges to the fixed point $Q_\alpha^{\pi_{\text{new}}}$ by Lemma 13.4. ■

Theorem 13.6 Convergence of soft policy iteration

In a finite MDP with bounded rewards and $\alpha > 0$, let π_0 be any policy and π_{k+1} the Boltzmann policy of $Q_\alpha^{\pi_k}$. Then $Q_\alpha^{\pi_k}$ increases with k, and

$$\left\|Q_\alpha^* - Q_\alpha^{\pi_k}\right\|_\infty \le \gamma^k \left\|Q_\alpha^* - Q_\alpha^{\pi_0}\right\|_\infty,$$

where Q_α^* is the unique solution of the soft Bellman optimality equation (13.7). Consequently $\pi_k \to \pi_\alpha^*$, and $Q_\alpha^\pi \le Q_\alpha^*$ for every policy π.

Proof. Let $(\mathcal{T}_\alpha Q)(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}[\alpha \log \sum_{a'} e^{Q(s',a')/\alpha}]$, the right-hand side of (13.7). The soft maximum is monotone and satisfies $\alpha \log \sum_a e^{(x_a+c)/\alpha} = \alpha \log \sum_a e^{x_a/\alpha} + c$ for every constant c. If $\|Q - Q'\|_\infty = \delta$, then $Q \le Q' + \delta$, so the soft maximum of $Q(s',\cdot)$ is at most that of $Q'(s',\cdot)$ plus δ, and symmetrically. Hence $\mathcal{T}_\alpha$ is monotone and a γ-contraction, with a unique fixed point Q^*_α (Banach's theorem, as in Chapter 4).

Two facts link $\mathcal{T}_\alpha$ to policies. First, for every policy π, $Q^\pi_\alpha = \mathcal{T}^\pi_\alpha Q^\pi_\alpha \le \mathcal{T}_\alpha Q^\pi_\alpha$, because by Proposition 13.3 the soft maximum is at least the value of any particular distribution; iterating the monotone $\mathcal{T}_\alpha$ gives $Q^\pi_\alpha \le \mathcal{T}^n_\alpha Q^\pi_\alpha \to Q^*_\alpha$. Second, $\mathcal{T}_\alpha Q^{\pi_k}_\alpha = \mathcal{T}^{\pi_{k+1}}_\alpha Q^{\pi_k}_\alpha$, because π_{k+1} attains the soft maximum in every state. With Lemma 13.5,

$$Q^{\pi_k}_\alpha \le \mathcal{T}_\alpha Q^{\pi_k}_\alpha = \mathcal{T}^{\pi_{k+1}}_\alpha Q^{\pi_k}_\alpha \le Q^{\pi_{k+1}}_\alpha \le Q^*_\alpha.$$

Therefore $0 \le Q^*_\alpha - Q^{\pi_{k+1}}_\alpha \le \mathcal{T}_\alpha Q^*_\alpha - \mathcal{T}_\alpha Q^{\pi_k}_\alpha$, whose sup norm is at most $\gamma \|Q^*_\alpha - Q^{\pi_k}_\alpha\|_\infty$; induction gives the bound. The Boltzmann map is continuous, so $\pi_{k+1} \to \pi^*_\alpha$, the Boltzmann policy of Q^*_α. Finally, $Q^{\pi^*_\alpha}_\alpha = Q^*_\alpha$, because $Q^*_\alpha = \mathcal{T}_\alpha Q^*_\alpha = \mathcal{T}^{\pi^*_\alpha}_\alpha Q^*_\alpha$, so Q^*_α is the fixed point of $\mathcal{T}^{\pi^*_\alpha}_\alpha$. Together with $Q^\pi_\alpha \le Q^*_\alpha$ for all π, this shows that π^*_α maximizes the soft objective from every state. ■

On the Trail, soft policy iteration from the uniform policy reaches the soft-optimal policy in two iterations ($\pi(\mathsf{go} \mid \mathsf{Start})$ goes 0.5, 0.895, 0.917), because the Trail has only two decisions to settle, from the end backward. Haarnoja, Zhou, Abbeel, et al. (2018) prove a version of Theorem 13.6 for a restricted class of policies, such as Gaussians, in which the Boltzmann policy is replaced by the policy of the class closest to it in KL divergence; improvement still holds (Exercise 13.13). Chapter 14 analyzes the same iteration with a step size, as entropy-regularized natural policy gradient (Theorem 14.7).

13.6 Soft actor-critic (SAC)

Soft actor-critic (Haarnoja, Zhou, Abbeel, et al. 2018; Haarnoja, Zhou, Hartikainen, et al. 2018) turns soft policy iteration into a deep RL algorithm. Soft policy evaluation becomes a critic trained by regression on soft Bellman targets from a replay buffer. Soft policy improvement becomes an actor trained to move toward the Boltzmann policy of the critic. The actor is stochastic, so SAC explores by sampling from it, and no noise needs to be added.

The critics. SAC uses two critics and their target copies, as TD3 does, with the target taken from the soft Bellman equation (13.6):

$$y = r + \gamma(1-d)\Big(\min_{j=1,2} Q_{\bar{w}_j}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}' \mid s')\Big), \qquad \tilde{a}' \sim \pi_\theta(\cdot \mid s'). \tag{13.8}$$

The next action $\tilde{a}'$ is sampled fresh from the *current* policy, not taken from the buffer, so the target estimates the soft value of the current policy from off-policy data, in the same way that DDPG's target uses the current actor's choice. One sample replaces the expectation over a', and each critic minimizes its mean squared error to y over the buffer.

The actor. Soft policy improvement replaces the policy by the Boltzmann policy, proportional to $\exp(Q(s,\cdot)/\alpha)$ with normalizer $Z(s)$, which a Gaussian network cannot represent exactly. SAC

therefore moves the policy toward it in KL divergence: it minimizes

$$\mathbb{E}_{s\sim\mathcal{D}}\Big[D_{\mathrm{KL}}\big(\pi_\theta(\cdot\mid s)\,\|\,e^{Q(s,\cdot)/\alpha}/Z(s)\big)\Big].$$

Expanding the KL divergence and multiplying by α (the normalizer $Z(s)$ does not depend on θ) gives the actor loss

$$L(\theta)=\mathbb{E}_{s\sim\mathcal{D}}\,\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\Big[\alpha\log\pi_\theta(a\mid s)-\min_{j=1,2}Q_{w_j}(s,a)\Big]. \tag{13.9}$$

In words: prefer actions the critics rate highly, but pay α for every nat of certainty.

The reparameterization trick. The expectation in (13.9) is over actions drawn from the policy being optimized. The score-function trick of Chapter 11 ((11.4)) could differentiate it, but SAC uses a lower-variance estimator. It writes the random action as a deterministic function of θ and independent noise, $a=f_\theta(\xi;s)$ with $\xi\sim\mathcal{N}(0,I)$, so that

$$\nabla_\theta L=\mathbb{E}_{s\sim\mathcal{D}}\,\mathbb{E}_{\xi}\Big[\nabla_\theta\Big(\alpha\log\pi_\theta\big(f_\theta(\xi;s)\mid s\big)-\min_j Q_{w_j}\big(s,f_\theta(\xi;s)\big)\Big)\Big].$$

The gradient now flows through the action into the critic, as in the deterministic policy gradient: SAC's actor also climbs the critic, but with a noisy action and an entropy term. This *reparameterization* (or pathwise) gradient is a staple of variational inference (Kingma and Welling 2014) and was used for policies by Heess et al. (2015). When it applies, its variance is usually far lower than that of the score-function estimator; in the example of Exercise 13.12 the ratio is almost 3,000.

13.6.1 Squashed Gaussian policies

Pendulum's torque, like most actuators, is bounded. A Gaussian policy can propose any real number, and clipping its samples would put a probability mass at the bounds that the Gaussian density does not account for. SAC instead *squashes* a Gaussian sample through the hyperbolic tangent. The actor network outputs a mean $m_\theta(s)$ and a log standard deviation $\log\sigma_\theta(s)$ for each action dimension, and the action is

$$u=m_\theta(s)+\sigma_\theta(s)\odot\xi,\qquad a=\tanh(u),\qquad \xi\sim\mathcal{N}(0,I),$$

with $\odot$ the componentwise product; the action $a\in(-1,1)^m$ is then scaled to the environment's range. The density of a follows from the change-of-variables formula. Because tanh acts on each coordinate separately, its Jacobian is diagonal with entries $1-\tanh^2(u_j)$, and

$$\log\pi_\theta(a\mid s)=\log\mathcal{N}\big(u;\,m_\theta(s),\mathrm{diag}(\sigma_\theta(s)^2)\big)-\sum_{j=1}^{m}\log\big(1-\tanh^2(u_j)\big). \tag{13.10}$$

For large $|u_j|$, $1-\tanh^2(u_j)$ underflows to zero in floating point; the identity $\log(1-\tanh^2 u)=2\big(\log 2-u-\log(1+e^{-2u})\big)$ computes it stably (Exercise 13.11). With a fixed ξ, the derivatives that the actor needs are simple. In one dimension, $\partial\log\pi/\partial m=2a$ and $\partial\log\pi/\partial\log\sigma=-1+2a\sigma\xi$, while $\partial a/\partial m=1-a^2$ and $\partial a/\partial\log\sigma=(1-a^2)\sigma\xi$ carry the critic's gradient $\nabla_a Q$ back to the

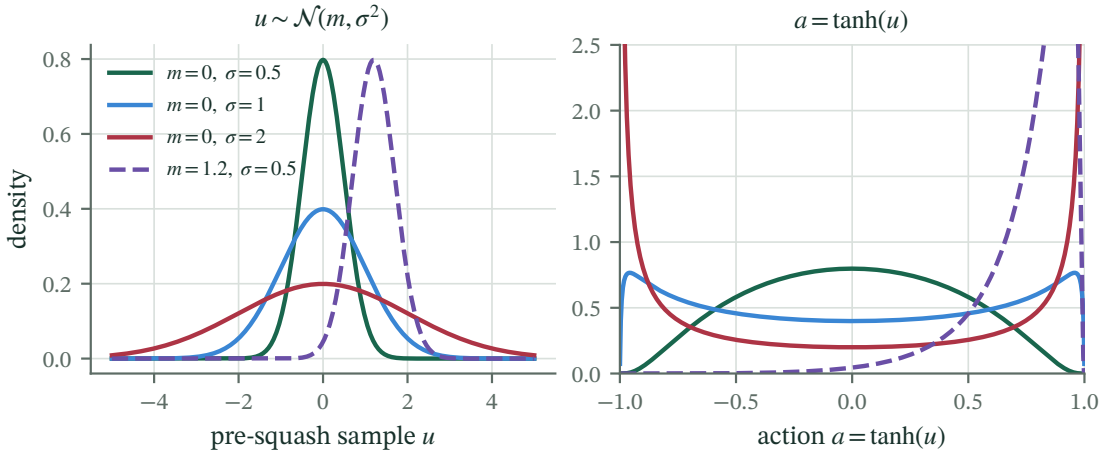


Figure 13.4. The tanh-squashed Gaussian. Left: densities of the pre-squash sample $u \sim \mathcal{N}(m, \sigma^2)$. Right: densities of the action $a = \tanh(u)$ from (13.10) (the vertical axis is cut at 2.5). A narrow Gaussian ($\sigma = 0.5$) stays bell-shaped; $\sigma = 1$ gives an almost flat density; a wide Gaussian ($\sigma = 2$) piles its mass against the bounds, where the density diverges. Shifting the mean toward a bound (dashed) also squeezes the mass against that bound.

network's outputs. The library's `SAC.actor_loss_and_grad` implements exactly these formulas, and a test checks them against finite differences.

Figure 13.4 holds a lesson for exploration: a wide Gaussian does not mean widely spread actions. With $\sigma = 2$, 36% of the squashed actions have $|a| > 0.95$, so the policy mostly bangs between the bounds. Squashing also lowers the entropy as the mean approaches a bound, which matters for the temperature, as we see next.

PITFALL

Forgetting the correction term in (13.10) is a common bug, and a silent one: the code still runs and still learns something. But the uncorrected expression is not a log-density (for $m = 0.4$ and $\sigma = 0.5$ the corresponding "density" integrates to 0.75 over the actions, not 1), so the entropy term and the temperature adjustment measure the wrong quantity. A related bug is to evaluate a trained SAC agent with sampled actions; for evaluation, use the deterministic action $\tanh(m_\theta(s))$.

13.6.2 Automatic temperature tuning

The temperature α is SAC's most sensitive hyperparameter, and its best value depends on the scale of the rewards, which changes from task to task and, as values sharpen, during training. Haarnoja, Zhou, Hartikainen, et al. (2018) proposed to choose a *target entropy* $\bar{\mathcal{H}}$ instead, and to adjust α automatically. The problem becomes a constrained one: maximize the expected return subject to the policy's entropy staying at least $\bar{\mathcal{H}}$ on average, $\mathbb{E}\big[-\log \pi(a_t \mid s_t)\big] \geq \bar{\mathcal{H}}$. The temperature plays the role of the Lagrange multiplier of this constraint, and it is adjusted by

gradient descent on the dual objective

$$J(\alpha) = \mathbb{E}_{s\sim\mathcal{D},\,a\sim\pi_\theta(\cdot|s)}\big[-\alpha \log \pi_\theta(a \mid s) - \alpha\bar{\mathcal{H}}\big], \qquad \frac{\partial J}{\partial \alpha} = \underbrace{\mathbb{E}\big[-\log \pi_\theta(a \mid s)\big]}_{\text{entropy}} - \bar{\mathcal{H}}. \tag{13.11}$$

If the policy's entropy is above the target, the derivative is positive and a descent step lowers α, which lets the policy become more decisive; if the entropy is below the target, α rises and the entropy term pushes the policy to spread out. Implementations optimize $\log\alpha$ rather than α, so that the temperature stays positive. The usual target is $\bar{\mathcal{H}} = -m$, minus the number of action dimensions, a heuristic of Haarnoja, Zhou, Hartikainen, et al. (2018); for one dimension it corresponds to a Gaussian with $\sigma \approx 0.09$ (Exercise 13.7). This dual-gradient reading is exact for the convex problem of choosing a policy for fixed values; in deep RL it is a heuristic that works well.

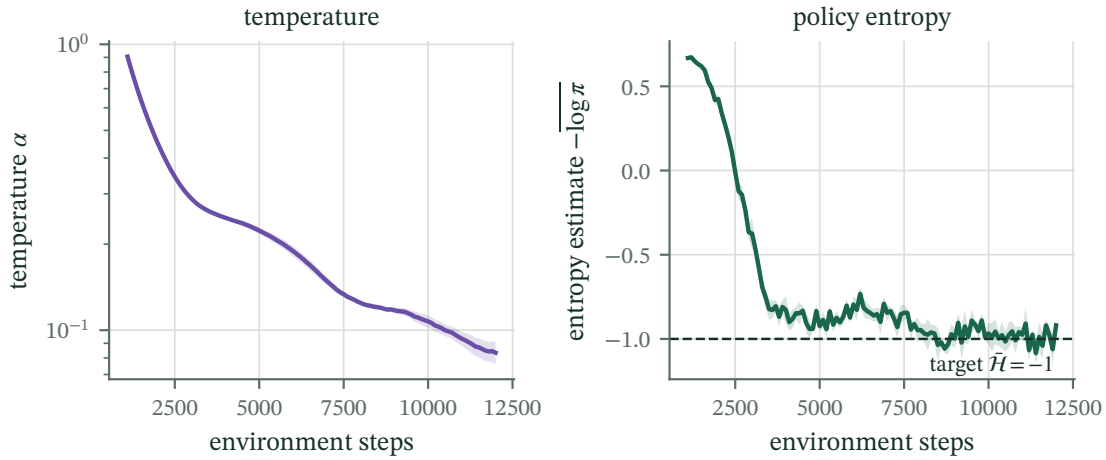


Figure 13.5. SAC's automatic temperature on Pendulum (five seeds; the band is ±1 standard error). Left: the temperature, on a log scale, falls from its initial value 1 as the critic's values sharpen. Right: the entropy of the policy, estimated as $-\overline{\log \pi}$ over 256 buffer states, drops from about 0.7 (a nearly uniform squashed policy) to the target −1 within about 4,000 steps and then tracks it.

Figure 13.5 shows the tuning at work: the mean temperature over five seeds is 0.45 after 2,000 steps, 0.19 after 6,000 and 0.08 after 12,000, while the entropy estimate averages −0.99 over the last 2,000 steps, against the target −1.

Algorithm 13.3 Soft actor-critic (SAC) with automatic temperature

Input: discount γ, averaging rate τ, step sizes, batch size B, target entropy $\bar{\mathcal{H}}$ (default $-m$), initial temperature α.

1. Initialize the squashed Gaussian actor π_θ and two critics Q_{w_1}, Q_{w_2}; set $\bar{w}_j \leftarrow w_j$; empty $\mathcal{D}$.
2. For each environment step:
   1. In state s, sample $a \sim \pi_\theta(\cdot \mid s)$; act; observe r, s', d; store (s, a, r, s', d) in $\mathcal{D}$.

2. Sample a minibatch from $\mathcal{D}$.
3. For each transition, sample $\tilde{a}'_i \sim \pi_\theta(\cdot \mid s'_i)$ and set $y_i \leftarrow r_i + \gamma(1-d_i)\big(\min_j Q_{\bar{w}_j}(s'_i, \tilde{a}'_i) - \alpha \log \pi_\theta(\tilde{a}'_i \mid s'_i)\big)$.
4. Update both critics by a gradient step on $\frac{1}{2B}\sum_i (Q_{w_j}(s_i, a_i) - y_i)^2$.
5. Sample $\xi_i \sim \mathcal{N}(0, I)$, set $a_i^\theta \leftarrow \tanh(m_\theta(s_i) + \sigma_\theta(s_i) \odot \xi_i)$, and take a gradient step on $\frac{1}{B}\sum_i \big[\alpha \log \pi_\theta(a_i^\theta \mid s_i) - \min_j Q_{w_j}(s_i, a_i^\theta)\big]$ through a_i^θ.
6. Temperature: $\log \alpha \leftarrow \log \alpha - \eta_\alpha \alpha\big(\frac{1}{B}\sum_i [-\log \pi_\theta(a_i^\theta \mid s_i)] - \bar{\mathcal{H}}\big)$.
7. Targets: $\bar{w}_j \leftarrow (1-\tau)\bar{w}_j + \tau w_j$.

Output: the policy; act with $\tanh(m_\theta(s))$ for evaluation.

SAC has no target actor: the target (13.8) samples from the current policy, whose randomness already smooths the target, much as TD3's target noise does. It updates the actor at every step, and it uses the minimum of the two critics both in the target and in the actor loss.

13.7 Pendulum with DDPG, TD3 and SAC

The library module `rlcourse.continuous` implements Algorithms 13.1, 13.2 and 13.3 in plain NumPy, with the settings of Table 13.1. They are close to the published defaults, scaled down for a laptop: smaller networks, a larger Adam step size, and a buffer that keeps every transition of the short runs.

Table 13.1. Settings for Pendulum in `rlcourse.continuous` (actions normalized to $[-1, 1]$; one gradient update per environment step after the warm-up).

Setting	DDPG	TD3	SAC
Networks	2 hidden layers, 64 ReLU	same, 2 critics	same, 2 critics
Actor output	tanh (deterministic)	tanh (deterministic)	mean, $\log \sigma$; squashed
Adam step size	10^{-3}	10^{-3}	10^{-3} (also for $\log \alpha$)
Batch, γ, τ	128, 0.99, 0.005	same	same
Exploration	Gaussian, $\sigma = 0.1$	Gaussian, $\sigma = 0.1$	samples from π_θ
Specific	none	$\tilde{\sigma} = 0.2$, $c = 0.5$, $k = 2$	$\alpha = 1$ at start, $\bar{\mathcal{H}} = -1$
Warm-up	1,000 random steps	same	same

Figure 13.6 shows five seeds of each method over 12,000 environment steps, 60 episodes. Every 1,000 steps the deterministic policy is evaluated from the same 10 start states. All three methods swing the pendulum up and hold it by about 6,000 steps, and all end near -140, the return of a good policy from these start states (final means -141 for DDPG, -135 for TD3, -138 for SAC).

They differ in the path. DDPG and SAC learn fastest, while TD3 is slower at first, because its actor is updated only every second step and its targets are smoothed. DDPG is the least stable: averaged over the last three evaluations its return is −210, against −157 for TD3 and −138 for SAC, because some of its seeds temporarily lose the good policy they had found.

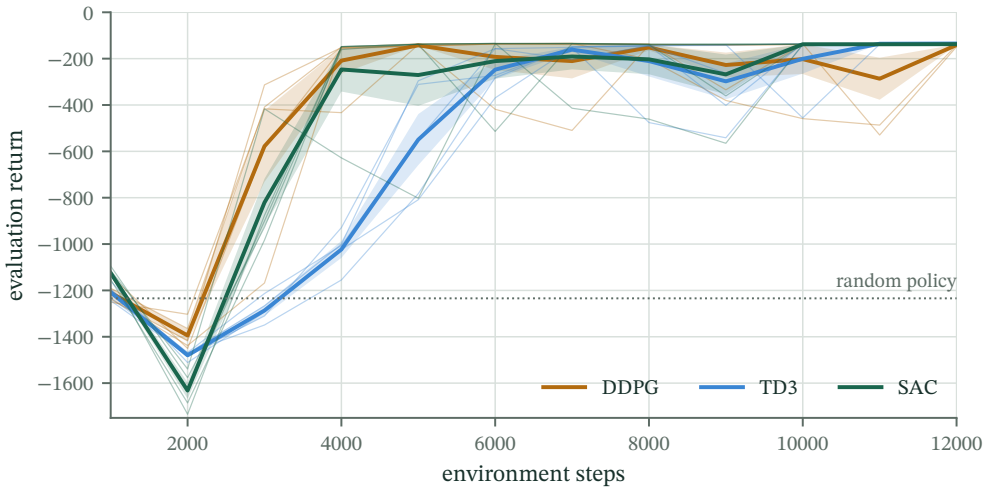


Figure 13.6. DDPG, TD3 and SAC on Pendulum with the NumPy library: evaluation return of the deterministic policy (average over 10 fixed start states) against environment steps. Thick lines are means over five seeds with a band of ±1 standard error; thin lines are individual seeds. The dotted line is the average return of a uniformly random policy. All three solve the task in about 6,000 steps; DDPG's seeds dip most often after solving it.

A few things are worth knowing before you run these methods yourself. Results vary across seeds, so look at several; one good curve proves little (Section 23.4). The dip at 2,000 steps, just after updates begin, is typical: the actor first pushes toward whichever torque the young critic happens to favor. When a run fails, the critic is the usual suspect, and three diagnostics locate most problems. Compare the critic's estimates with Monte Carlo returns of the current policy, as in Figure 13.2; a critic that keeps rising above them is overestimating. Watch whether the actor's outputs saturate at ±1 for all states, a sign that it is chasing a critic error at the bound. For SAC, watch α and the entropy: an entropy far from its target for long means the temperature step size is too small, and a temperature collapsing to zero early means the critic's values are growing out of control. Finally, report the deterministic policy's return, measured separately, rather than the returns of the exploring policy during training.

LAB

Lab 13.1: Continuous control with DDPG, TD3 and SAC (`labs/ch13_continuous.py`, about 45 minutes; the script runs in about two minutes). Run `python3 labs/ch13_continuous.py`. You will check the deterministic policy gradient on Example 13.1 and on a linear-quadratic problem, verify the squashed-Gaussian correction, compute the soft Trail of Example 13.2, train DDPG and TD3 on Pendulum while measuring the overestimation of their critics, and train SAC while watching its temperature. The "Going further" ideas ablate TD3 and raise the update-to-data ratio.

13.8 Comparing the families

We now have two families of deep RL methods for continuous control: on-policy methods such as PPO (Algorithm 12.2) and off-policy actor-critic methods such as TD3 and SAC. Which to choose depends less on the algorithms' elegance than on what limits you.

Sample efficiency. An off-policy method learns from every transition many times, and PPO uses each batch for a few epochs and discards it. On the standard simulated locomotion benchmarks, SAC and TD3 reported higher returns than PPO for the same number of environment steps (Haarnoja, Zhou, Abbeel, et al. 2018; Fujimoto et al. 2018). When each step is expensive (a real robot, a slow simulator, a human in the loop) this is decisive, and off-policy methods have learned on real robots in hours (Haarnoja, Zhou, Hartikainen, et al. 2018).

Wall-clock time and parallel simulation. When steps are cheap, the comparison flips. PPO needs no replay buffer and makes a few large-batch updates per batch of experience, so it scales well to thousands of parallel simulated environments; with GPU-based simulators, legged-robot walking policies have been trained in minutes of wall-clock time (Rudin et al. 2022). An off-policy method makes one or more gradient updates per environment step, which caps how fast it can consume experience.

Stability and tuning. PPO's defaults transfer well across tasks, though its implementation details matter (Section 12.7; Engstrom et al. (2020), Andrychowicz et al. (2021)). Among the off-policy methods, DDPG is the most brittle, TD3 and SAC are much more stable, and SAC's automatic temperature removes its most sensitive setting. All of them depend on the critic, so a critic failure, such as overestimation or divergence, can wreck the policy.

Action spaces, data and exploration. PPO handles discrete and continuous actions alike. DDPG and TD3 need continuous actions, because they differentiate the critic with respect to the action; SAC has discrete-action variants (Christodoulou 2019). Off-policy methods can learn from any stored data, including demonstrations and old experiments, which is the starting point of offline RL (Chapter 17, where TD3 with an added behavior-cloning term turns out to be a strong offline method). SAC explores through its objective; TD3 relies on the noise added to its actions.

Table 13.2. A rough guide to choosing between on-policy and off-policy methods. These are tendencies reported across many studies, not laws; when in doubt, try both with several seeds.

Question	PPO (on-policy)	SAC or TD3 (off-policy)
Is experience expensive?	no: fast, parallel simulators	yes: robots, slow simulators
What limits you?	wall-clock time, tuning effort	environment steps
Action space	discrete or continuous	continuous (SAC: also discrete variants)
Memory	no replay buffer	buffer of up to millions of transitions
Exploration	policy noise, entropy bonus	SAC: entropy objective; TD3: action noise

Question	PPO (on-policy)	SAC or TD3 (off-policy)
Reusing old data or demonstrations	not directly	naturally, through replay

HISTORICAL NOTE

The deterministic policy gradient theorem was proved by Silver et al. (2014), and DDPG brought it to deep networks two years later (Lillicrap et al. 2016). In 2018 two groups independently addressed DDPG's weaknesses: Fujimoto et al. (2018) with TD3, by controlling overestimation, and Haarnoja, Zhou, Abbeel, et al. (2018) with SAC, by combining an off-policy critic with the maximum-entropy framework of Ziebart (2010) and soft Q-learning (Haarnoja et al. 2017). Both remain standard baselines.

13.9 Further developments ●●

TD3 and SAC were the end of one line of work and the starting point of several others. This section surveys the main ones; the papers cited are the best entry points.

Maximum a posteriori policy optimization. MPO (Abdolmaleki et al. 2018) casts policy improvement as expectation maximization. In the E-step it builds a nonparametric improved policy $q(a \mid s) \propto \pi_{\text{old}}(a \mid s) \exp(Q(s,a)/\eta)$, whose temperature η is chosen by solving a small dual problem so that q stays within a fixed KL divergence of the old policy. In the M-step it fits the parametric policy to q by weighted maximum likelihood, again under a KL trust region. The E-step is the Boltzmann improvement of soft policy iteration, anchored to the old policy as in relative entropy policy search (Peters et al. 2010); the hard KL constraints replace SAC's temperature and are reported to make MPO robust across tasks.

Distributional critics. The distributional methods of Section 10.8 carry over to actor-critic methods. D4PG (Barth-Maron et al. 2018) gave DDPG a categorical return distribution as critic, together with n-step returns, prioritized replay and many parallel actors. TQC (Kuznetsov et al. 2020) controls overestimation with distributions rather than with a minimum: it trains several critics that each predict quantiles of the return, pools their quantiles, and drops the largest few before forming the target. The number of dropped quantiles tunes the bias continuously, from optimistic to pessimistic, instead of the all-or-nothing minimum of two critics.

More updates per step. The ratio of gradient updates to environment steps, the *update-to-data* (UTD) or replay ratio, is 1 in TD3 and SAC. Raising it should extract more from each transition, but naively it makes things worse: the critics overfit the buffer and their overestimation grows. REDQ (X. Chen et al. 2021) made high ratios work with an ensemble of critics, typically ten, and targets that take the minimum over a random pair of them; with a ratio of 20 it matched the sample efficiency of a model-based method on locomotion benchmarks. DroQ (Hiraoka et al. 2022) obtained similar results with two critics regularized by dropout and layer normalization. D'Oro et al. (2023) showed that periodically resetting the networks' parameters, while keeping the buffer, lets the replay ratio grow much further. Other recent work moves in different directions: CrossQ (Bhatt et al. 2024) removes target networks and uses batch normalization in the

critic; BRO (Nauman et al. 2024) scales the critic network up, with strong regularization and optimistic exploration; TD7 (Fujimoto et al. 2023) adds learned state-action embeddings to TD3. These reported gains come from benchmarks and are still being sorted out; Exercise 13.18 asks what is known.

Large-scale off-policy actor-critic. In IMPALA (Espeholt et al. 2018), many actors generate trajectories with slightly stale copies of the policy and one learner trains on them. The mismatch between the behavior policy b and the learner's policy π is corrected by *V-trace* targets with truncated importance ratios $\rho_t = \min(\bar{\rho}, \pi(a_t \mid s_t)/b(a_t \mid s_t))$ and $c_t = \min(\bar{c}, \pi(a_t \mid s_t)/b(a_t \mid s_t))$:

$$v_t = V(s_t) + \sum_{k=t}^{t+n-1} \gamma^{k-t} \Big(\prod_{i=t}^{k-1} c_i \Big) \rho_k \big(r_k + \gamma V(s_{k+1}) - V(s_k) \big).$$

The value network regresses on v_t and the policy follows $\rho_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)(r_t + \gamma v_{t+1} - V(s_t))$. With $\bar{\rho} \geq \bar{c}$, the truncation trades a small bias for bounded variance: the targets are those of a policy between π and b. The construction is a sibling of Retrace (Section 7.8); unlike the methods of this chapter, it corrects off-policy data with importance weights, which is affordable because the data are only slightly off-policy.

13.9.1 Soft Q-learning and policy gradients ●●●

Maximum-entropy RL blurs the line between value-based and policy-based methods. A soft Q-function defines a policy, its Boltzmann policy, and Schulman, Chen, et al. (2017) showed that soft Q-learning and entropy-regularized policy gradients are, in a precise sense, the same algorithm. The one-step case shows the mechanism. Let $Q_\theta(a)$ be a parameterized soft Q-function on a finite action set, and define $V_\theta = \alpha \log \sum_a e^{Q_\theta(a)/\alpha}$ and $\pi_\theta(a) = e^{(Q_\theta(a) - V_\theta)/\alpha}$, so that $Q_\theta(a) = V_\theta + \alpha \log \pi_\theta(a)$. Soft Q-learning on a sampled action $a \sim \pi_\theta$ with reward r takes a step along the negative gradient of $\frac{1}{2}(Q_\theta(a) - r)^2$, which is

$$\big(r - Q_\theta(a)\big) \nabla_\theta Q_\theta(a) = \alpha\, \delta\, \nabla_\theta \log \pi_\theta(a) + \delta\, \nabla_\theta V_\theta, \qquad \delta = r - \alpha \log \pi_\theta(a) - V_\theta.$$

The first term is α times a sampled policy gradient of the entropy-regularized objective $\mathbb{E}_{a \sim \pi_\theta}[r(a)] + \alpha \mathcal{H}(\pi_\theta)$, with the baseline V_θ; the second moves V_θ toward the entropy-augmented return. So the "value" update is a policy-gradient step plus a baseline update (Exercise 13.16). Schulman, Chen, et al. (2017) extend this to multi-step returns and relate it to natural policy gradients; path consistency learning (Nachum et al. 2017) and PGQL (O'Donoghue et al. 2017) build algorithms on related identities.

13.9.2 Reinforcement learning as inference ●●●

Why the logarithm of a sum of exponentials, and why the entropy? A probabilistic reading answers both (Levine 2018). Add to the MDP a binary "optimality" variable $\mathcal{O}_t$ at each step, with $\Pr(\mathcal{O}_t = 1 \mid s_t, a_t) = e^{r(s_t, a_t)}$ (rewards shifted to be nonpositive), and ask for the distribution of trajectories conditioned on $\mathcal{O}_t = 1$ at every step. Exact inference by backward messages yields log-sum-exp backups, but with a twist: the backup over next states is also soft,

$Q(s,a) = r(s,a) + \log \mathbb{E}_{s'}\left[e^{V(s')}\right]$, as if the agent could choose the outcome of chance. The resulting behavior is risk-seeking in stochastic environments. If instead we look for the best approximation to that posterior among distributions that keep the true dynamics and choose only a policy, the variational objective is exactly the maximum-entropy objective (13.5) with $\alpha = 1$, and the backup becomes the soft Bellman equation with an ordinary expectation over s'. The temperature corresponds to rescaling the rewards. This view connects SAC to maximum causal entropy (Ziebart 2010), to the EM structure of MPO, and to maximum-entropy inverse RL (Algorithm 18.3); Exercise 13.17 works through the optimistic backup.

RESEARCH NOTE

Open questions: How should the temperature or target entropy be set in principle, for example from the cost of exploration? Why do high replay ratios fail without resets or ensembles, and is the cause overfitting, overestimation or a loss of plasticity (Section 24.5)? Can off-policy critics be made as reliable as on-policy value estimates at scale?

13.10 Summary

- With continuous actions, the $\max_a Q(s,a)$ of Q-learning is an optimization problem. Off-policy actor-critic methods learn a critic from replay and an actor that outputs the action the critic rates highest.
- The deterministic policy gradient theorem says that the gradient of a deterministic policy's return is the critic's action gradient, passed through the actor by the chain rule and averaged over the discounted visitation distribution. On a one-step problem it can be computed by hand.
- DDPG combines the deterministic policy gradient with replay, Polyak-averaged target networks and exploration noise. It learns fast but is brittle, partly because an actor that climbs a noisy critic makes the critic overestimate.
- TD3 adds clipped double-Q targets, which trade overestimation for slight underestimation, target policy smoothing and delayed actor updates. On Pendulum, DDPG's critic settled about 13 above the true values and TD3's about 4 below.
- Maximum-entropy RL adds α times the policy's entropy to every reward. Its values obey the soft Bellman equation, its optimal policy is the Boltzmann policy of the soft Q-function, and (●●) soft policy iteration converges at rate γ. On the Trail with $\alpha = 1$, the soft-optimal policy goes on with probabilities 0.917 and 0.832.
- SAC trains two soft critics and a tanh-squashed Gaussian actor with the reparameterization trick, corrects the log-probability for the squashing, and tunes α to keep the policy's entropy at a target.
- On Pendulum, all three methods solve the task in about 6,000 steps with the NumPy library; SAC was the most stable and DDPG the least.
- Off-policy methods win when experience is expensive; on-policy methods often win when simulation is cheap and massively parallel.

The next chapter steps back from algorithms to theory: why gradient methods find optimal policies at all, how fast, and what entropy regularization buys.

13.11 Further Reading

The original papers are the best introduction to each method. Silver et al. (2014) derive the deterministic policy gradient and its off-policy form, and Lillicrap et al. (2016) describe DDPG and its engineering. Fujimoto et al. (2018) is a model of clear experimental analysis: it measures overestimation in actor-critic methods before fixing it. For SAC, read Haarnoja, Zhou, Abbeel, et al. (2018) for the derivation from soft policy iteration and Haarnoja, Zhou, Hartikainen, et al. (2018) for automatic temperature tuning and experiments on real robots.

For maximum-entropy RL, Ziebart (2010) is the foundational thesis, Haarnoja et al. (2017) introduced soft Q-learning with expressive energy-based policies, and Levine (2018) is a readable tutorial on the inference view. Eysenbach and Levine (2022) prove that maximum-entropy policies are robust to certain perturbations of rewards and dynamics. The reparameterization trick is explained in Kingma and Welling (2014), and Heess et al. (2015) develop value gradients for stochastic policies. For value-based alternatives in continuous action spaces, see Gu et al. (2016) and Kalashnikov et al. (2018).

On reproducibility, Henderson et al. (2018) remains a sobering read: it shows how much the results of DDPG and other methods depend on seeds, implementations and hyperparameters. Andrychowicz et al. (2021) and Engstrom et al. (2020) do the same for on-policy methods, which is useful background for Section 13.8.

(●●●) Recent work on sample efficiency is moving fast; start with REDQ (X. Chen et al. 2021), DroQ (Hiraoka et al. 2022) and the study of replay ratios and resets by D'Oro et al. (2023), then see CrossQ (Bhatt et al. 2024), BRO (Nauman et al. 2024) and TD7 (Fujimoto et al. 2023). For off-policy correction at scale, read IMPALA (Espeholt et al. 2018); for the links between value-based and policy-based methods under entropy regularization, Schulman, Chen, et al. (2017), Nachum et al. (2017) and O'Donoghue et al. (2017).

13.12 Exercises

Exercise 13.1 ● The deterministic policy gradient by hand

A one-step problem has states $s = 1$ and $s = 2$, each with probability $\frac{1}{2}$, real actions, and reward $r(s, a) = -(a - 3s)^2$; the episode ends after one step. The policy is $\mu_\theta(s) = \theta s$. (a) For $\theta = 1$, compute $\nabla_a Q(s, a)$ at $a = \mu_\theta(s)$ in each state and the deterministic policy gradient. (b) Compute $J(\theta)$ in closed form and check (a) by differentiating. (c) Take one gradient-ascent step with step size 0.1. What are θ and $J(\theta)$ before and after, and what is the optimal θ?

Exercise 13.2 ● The cost of discretizing

A robot arm has 7 joints, and each joint's torque is discretized into 10 levels. (a) How many discrete actions does a DQN need, and how many output units? (b) How many critic evaluations would computing one target $\max_{a'} Q(s', a')$ by enumeration take, and for a minibatch of 256 transitions? (c) Pendulum has one action dimension. Why might DQN with 21 torque levels

work well there? (d) Name two ways other than an actor to avoid enumerating the actions, and the price of each.

Exercise 13.3 • Polyak averaging

With soft target updates $\bar{w} \leftarrow (1-\tau)\bar{w} + \tau w$ and $\tau = 0.005$: (a) After how many updates has the weight of the initial target parameters fallen below one half? (b) Show that the target equals a weighted average of the online parameters of past updates (plus a decaying term from the initialization), and compute the average age of the parameters in this average. (c) What happens to learning with $\tau = 1$, and with τ very close to 0?

Exercise 13.4 • Which way does the minimum lean?

Let X_1 and X_2 be independent estimates of the same value μ, each normally distributed with mean μ and standard deviation σ. (a) Show that $\mathbb{E}[\min(X_1, X_2)] = \mu - \sigma/\sqrt{\pi}$ and $\mathbb{E}[\max(X_1, X_2)] = \mu + \sigma/\sqrt{\pi}$. (Hint: $\min(x, y) = \frac{1}{2}(x+y) - \frac{1}{2}|x-y|$.) (b) Evaluate both for $\sigma = 1$. (c) Explain why TD3 accepts the downward bias of the minimum rather than an upward bias, using the behavior of the actor.

Exercise 13.5 • Target policy smoothing against a spike

In some state s' the true values are $Q(s', a) = -a^2$ on $[-1, 1]$, but the critic has learned a spurious narrow spike: $Q_w(s', a) = -a^2 + 2\max(0, 1 - |a - 0.5|/0.02)$. (a) Which action would a deterministic actor that maximizes $Q_w(s', \cdot)$ choose, and what target value would DDPG use? (b) TD3 smooths the target with $\tilde{a}' = \mu(s') + \text{clip}(\tilde{\sigma}\xi, -0.5, 0.5)$, $\tilde{\sigma} = 0.2$. Estimate $\mathbb{E}[Q_w(s', \tilde{a}')]$ when the actor sits on the spike, $\mu(s') = 0.5$, and when it sits at the true optimum, $\mu(s') = 0$. (You may use that $\mathbb{E}[\text{clip}(\tilde{\sigma}\xi, -0.5, 0.5)^2] \approx 0.039$, and treat the spike as a triangle of area 0.04.) (c) What does the comparison mean for the actor?

Exercise 13.6 • The soft value of a two-action state

A state has two actions with soft action values $Q(\text{go}) = 3.6$ and $Q(\text{exit}) = 2$ (the Middle state of the Trail). (a) For $\alpha = 1$, compute the soft-optimal policy, its expected action value $\mathbb{E}_\pi[Q]$, its entropy, and the soft value, and check that the soft value equals $\log(e^{3.6} + e^2)$. (b) Repeat for $\alpha = 0.5$. (c) Show that for any two actions the soft-optimal probability of the first is $\sigma((Q_1 - Q_2)/\alpha)$, with σ the logistic function.

Exercise 13.7 • Reading the temperature update

(a) From (13.11), show that one gradient step lowers α exactly when the policy's entropy is above the target $\bar{\mathcal{H}}$. (b) Which standard deviation does a one-dimensional Gaussian need for its entropy to equal -1? (c) For a tanh-squashed Gaussian with that standard deviation, the entropy is about -1.01 when the pre-squash mean is 0, but about -2.71 when the mean

is 1.5. Explain the difference, and predict what automatic tuning does when the best action of a task lies at a bound of the action range.

Exercise 13.8 ● The time-limit bug

Consider a continuing task with reward -1 at every step, $\gamma = 0.99$, and episodes cut off after 200 steps; the state carries no information about time. (a) What is the true value of every state? (b) Suppose the cut-off is wrongly stored as termination. What return does the critic's target describe for a state visited k steps before the cut, for $k = 200$ and $k = 1$? (c) If the states are visited at uniformly random times within the episode, what single value does a critic that cannot see time learn on average? (d) Why is the damage different for an agent whose rewards are near zero at the end of each episode, as for a good Pendulum policy?

Exercise 13.9 ●● The deterministic policy gradient on a linear-quadratic problem

A scalar system evolves as $x_{t+1} = ax_t + bu_t$, with reward $-(qx_t^2 + \rho u_t^2)$ at each step, discount γ, and the linear policy $u_t = \theta x_t$. Let $c = a + b\theta$ and assume $\gamma c^2 < 1$. (a) Show that $V^\theta(x) = -P(\theta)x^2$ with $P(\theta) = (q + \rho\theta^2)/(1 - \gamma c^2)$, and write $Q^\theta(x, u)$. (b) Compute $\nabla_\theta J$ directly from $J(\theta) = -P(\theta)\mathbb{E}[x_0^2]$. (c) Compute the deterministic policy gradient (13.1) and show that it agrees with (b). (d) With $a = b = q = \rho = 1$, $\gamma = 0.9$ and $\mathbb{E}[x_0^2] = 1$, find the optimal gain from the discounted Riccati equation $P = q + \gamma a^2 P - \gamma^2 a^2 b^2 P^2/(\rho + \gamma b^2 P)$, $\theta^* = -\gamma abP/(\rho + \gamma b^2 P)$, and evaluate the gradient at $\theta = -0.5$.

Exercise 13.10 ●● The soft maximum and its limits

(a) Prove Proposition 13.3 with a Lagrange multiplier for the constraint $\sum_a p(a) = 1$, as an alternative to the KL argument. (b) Show that $\max_a q(a) \le \alpha \log \sum_a e^{q(a)/\alpha} \le \max_a q(a) + \alpha \log |\mathcal{A}|$, and find the limits of the soft-optimal policy as $\alpha \to 0$ (consider ties) and as $\alpha \to \infty$. (c) Show that the gradient of the soft maximum with respect to the vector q is the soft-optimal policy.

Exercise 13.11 ●● Change of variables for the tanh squashing

(a) Derive (13.10) for one action dimension from the change-of-variables formula for densities, and then for m dimensions. (b) Prove the identity $\log(1 - \tanh^2 u) = 2(\log 2 - u - \log(1 + e^{-2u}))$ and explain why it is preferable numerically. (c) If the environment's action is $a_{\text{env}} = 2a$, as for Pendulum's torque, how does $\log \pi(a_{\text{env}} \mid s)$ differ from $\log \pi(a \mid s)$, and does the difference matter for SAC?

Exercise 13.12 ●● Reparameterization versus the score function

Let $a \sim \mathcal{N}(m, \sigma^2)$ and $f(a) = -(a - c)^2$; write $\delta = m - c$. (a) Show that the reparameterized estimator $g_{\text{rep}} = f'(m + \sigma\xi)$ and the score-function estimator $g_{\text{sf}} = f(a)(a - m)/\sigma^2$ are both unbiased for $\frac{d}{dm}\mathbb{E}[f(a)]$. (b) Show that $\text{Var}(g_{\text{rep}}) = 4\sigma^2$ and $\text{Var}(g_{\text{sf}}) = \delta^4/\sigma^2 + 14\delta^2 + 15\sigma^2$. (c)

Evaluate both for $\delta = 1$ and $\sigma = 0.1$. What happens to the score-function estimator as the policy becomes nearly deterministic? (d) Name a situation in which the reparameterized estimator cannot be used.

Exercise 13.13 ●● Soft policy improvement in a restricted class

Let Π be a set of policies that contains π_{old}, and let π_{new} minimize $D_{\text{KL}}\big(\pi(\cdot \mid s) \,\|\, e^{Q_\alpha^{\pi_{\text{old}}}(s,\cdot)/\alpha}/Z(s)\big)$ over $\pi \in \Pi$, in every state. Prove that $Q_\alpha^{\pi_{\text{new}}} \geq Q_\alpha^{\pi_{\text{old}}}$. Which step of the proof of Lemma 13.5 changes, and what does the argument no longer give?

Exercise 13.14 ●● Programming: TD3 ablations

Using `rlcourse.continuous.TD3`, train four variants on Pendulum for 12,000 steps with three seeds each: full TD3, without clipped double Q (`clipped_double_q=False`), without target smoothing (`target_noise=0.0`), and without the policy delay (`policy_delay=1`). Report how many steps each variant needs to reach an evaluation return above -300, its mean return over the evaluations from 4,000 to 8,000 steps, and its final returns. Which change matters on Pendulum, in which direction, and how much can three seeds tell you?

Exercise 13.15 ●● On-policy or off-policy?

For each situation, recommend PPO or SAC/TD3 and justify the choice with the considerations of Section 13.8: (a) a quadruped robot trained in a GPU simulator with 4,096 parallel environments; (b) the same robot fine-tuned on hardware, with an hour of operating time a day; (c) a recommendation system with discrete items and logged data from the current system; (d) a chemical process where each experiment takes a day and old experimental records exist. Then explain why PPO cannot simply reuse a replay buffer, referring to its clipped objective (12.6).

Exercise 13.16 ●●● Soft Q-learning is a policy gradient

(a) Verify the decomposition in Section 13.9.1: with $Q_\theta(a) = V_\theta + \alpha \log \pi_\theta(a)$, show that $(r - Q_\theta(a))\nabla_\theta Q_\theta(a) = \alpha\delta\nabla_\theta \log \pi_\theta(a) + \delta \nabla_\theta V_\theta$. (b) Show that $\mathbb{E}_{a\sim\pi_\theta}[\delta\, \nabla_\theta \log \pi_\theta(a)] = \nabla_\theta\big(\mathbb{E}_{a\sim\pi_\theta}[r(a)] + \alpha\mathcal{H}(\pi_\theta)\big)$ when r does not depend on θ. (c) What does the second term do? (d) Read Schulman, Chen, et al. (2017) and explain in a paragraph how the multi-step version uses n-step soft returns, and what the authors observe about Q-learning without target networks.

Exercise 13.17 ●●● Optimism in RL as inference

A state s has two actions. Action safe leads with certainty to a state worth $V = 0$; action gamble leads with probability 0.01 to a state worth $V = 5$ and otherwise to a state worth $V = -5$; both have reward 0. (a) Compare the two actions' values under the ordinary backup $\mathbb{E}_{s'}[V(s')]$ and under the "exact inference" backup $\log \mathbb{E}_{s'}[e^{V(s')}]$ of Section 13.9.2. (b) Explain why the second backup is called optimistic, and why it describes the posterior over trajectories given optimality

but not a policy the agent can carry out. (c) Summarize, following Levine (2018), how restricting the variational distribution to the true dynamics removes the optimism, and state the resulting objective.

Exercise 13.18 ●●● Research: high replay ratios and resets

Off-policy methods make one update per environment step. Survey what happens when this update-to-data ratio is raised to 10 or 20. (a) Explain at least two mechanisms by which naive high ratios hurt learning. (b) Describe how REDQ, DroQ and periodic resets each counter these mechanisms. (c) Propose a small experiment with `rlcourse.continuous` and Pendulum (the `updates_per_step` argument of `train`) that would separate overestimation from overfitting as causes of failure, and say what you would measure.

CHAPTER 14

The Theory of Policy Optimization

WHAT YOU WILL LEARN

- Explain in words why policy gradient methods find optimal policies of finite MDPs although their objective is not concave, and what can make them slow.
- State and prove the gradient domination lemma, and compute distribution mismatch coefficients on small examples (●●).
- Prove an $O(1/t)$ convergence rate for softmax policy gradient on bandits, and state the global convergence results for MDPs together with their lower bounds (●●).
- Show that natural policy gradient for softmax policies is a multiplicative-weights update, and prove its dimension-free $O(1/T)$ rate (●●).
- State the linear convergence of entropy-regularized natural policy gradient and bound the bias that regularization introduces (●●).
- Describe what changes with function approximation, in linear-quadratic control, in partially observed problems and in games, and name open problems (●●●).

The policy gradient methods of Chapter 11 and Chapter 12 are among the most widely used tools in reinforcement learning, from robot locomotion to the fine-tuning of language models. Yet the reason they work at all is not obvious. They climb the expected return $J(\theta)$ by gradient ascent, and J is not a concave function of the policy parameters. For general nonconvex functions, gradient methods are only guaranteed to reach a point where the gradient vanishes, which can be a poor local optimum, a saddle point, or a plateau where the gradient is tiny but the function is far from its maximum. Why, then, does policy gradient find the best policy of a finite MDP? And how fast?

This chapter answers these questions for the setting in which they are best understood: finite MDPs with exact gradients. The answers have a clear structure. The objective has no spurious local optima, because a policy that cannot be improved locally cannot be improved at all; the precise form of this statement is the *gradient domination* property. Plain softmax policy gradient therefore converges to an optimal policy, but its speed depends on constants that can be astronomically bad. Natural policy gradient removes most of that dependence, and entropy regularization makes convergence linear, at the price of a small, controllable bias. With function approximation, partial observability or several agents, these guarantees weaken or fail, and much remains open.

Section 14.1 tells this story in words, with the Trail as the example; it is written for every reader and can be read alone. The rest of the chapter is for the Graduate and Research tracks. Section 14.2 (●●) proves gradient domination, Section 14.3 (●●) analyzes softmax policy gradient,

Section 14.4 (●●) proves the rate of natural policy gradient and introduces policy mirror descent, and Section 14.5 (●●) treats entropy regularization. Section 14.6 and Section 14.7 (●●●) go beyond the tabular setting. The prerequisites are the policy gradient theorem (Theorem 11.1), the natural gradient (Section 12.4) and the performance difference lemma (Lemma 3.10).

14.1 The landscape of the objective

Policy gradient methods treat reinforcement learning as an optimization problem: choose the parameters θ of a policy to maximize the expected return $J(\theta)$. Whether gradient ascent finds the maximum depends on the shape of J, its *landscape*. If J were concave, like an upside-down bowl, every uphill path would lead to the top. It is not, and the Trail shows what can happen instead.

14.1.1 The Trail in logit space

On the Trail (Chapter 1) a policy is described by the probabilities p_0 and p_1 of go in Start and Middle. With $\gamma = 0.9$ its expected return is

$$J(p_0, p_1) = 1 + p_0\,(0.8 + 1.44\,p_1),$$

which is the formula of Chapter 1 with the numbers filled in. Policy gradient methods usually do not work with probabilities directly, because probabilities must stay between 0 and 1. They use *logits* (unconstrained real numbers) and turn them into probabilities with the sigmoid function, $p_i = \sigma(z_i) = 1/(1 + e^{-z_i})$; with more than two actions, the softmax function plays this role. In logit space the Trail's return is $J(z_0, z_1) = 1 + \sigma(z_0)\big(0.8 + 1.44\,\sigma(z_1)\big)$, and the left panel of Figure 14.1 shows it.

Three features of the picture recur throughout the chapter.

The function is not concave. Along the line $z_1 = 0$ the return is $1 + 1.52\,\sigma(z_0)$, an S-shaped curve. At $z_0 = -4, -2, 0$ it equals 1.027, 1.181 and 1.760. The middle value lies below the average of its neighbors, 1.394, which a concave function would never allow. In the probability parameters the return is not concave either: along the diagonal $p_0 = p_1 = t$ it is $1 + 0.8t + 1.44t^2$, a convex parabola.

There are plateaus. At $z = (-5, -5)$ the policy almost always exits at once, and $J = 1.005$, far below the optimum 3.24. Yet the gradient there has size about 0.005, nearly eighty times smaller than at the uniform policy $z = (0, 0)$, where it is $(0.38, 0.18)$. A gradient method started on a plateau moves very slowly, and in high dimensions a plateau can be hard to tell apart from a maximum.

The maximum is at infinity. The best policy, go and go, is deterministic, and a sigmoid reaches 1 only as its logit goes to infinity. So gradient ascent never arrives; it approaches. Near the optimum the gradient shrinks again, because the sigmoid flattens.

What the picture does *not* show is just as important: there are no spurious local maxima. The gradient is never zero, and every uphill path ends near the top-right corner. This is not luck. The Trail shares it with every finite MDP.

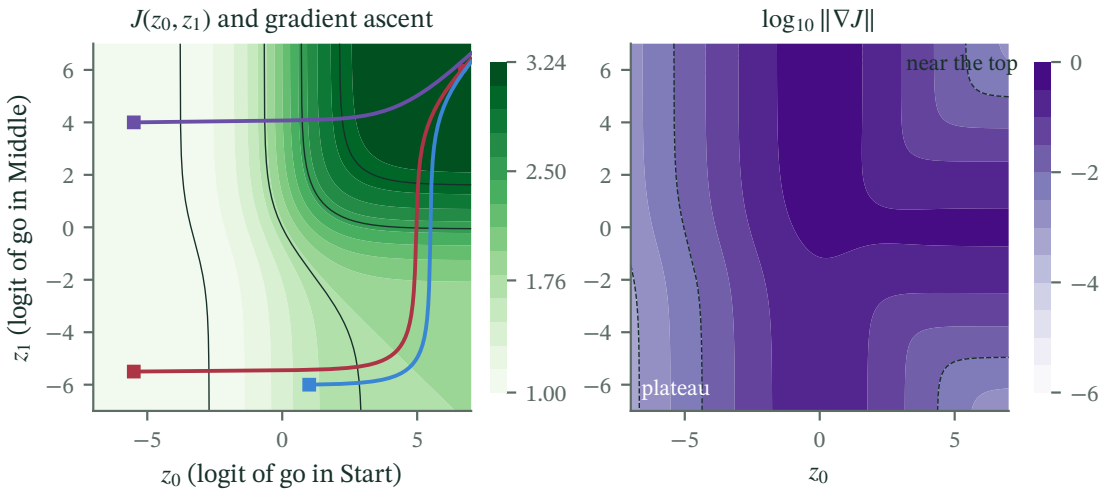


Figure 14.1. The Trail's expected return in logit space ($\gamma = 0.9$). Left: contours of $J(z_0, z_1)$, with three runs of gradient ascent from poor starting points (squares; dots every 400 steps). Every run ends near the top-right corner, where J approaches its supremum 3.24. Right: the size of the gradient on a log scale. Far from the optimum (left edge) the gradient is tiny although J is far from its best value: a plateau. Near the top it is small again, because the optimum lies at infinity.

14.1.2 What the theory says, in words

The results of this chapter can be summarized in five statements. Each is made precise, with its conditions, in the later sections.

1. *No local traps (gradient domination).* If a policy cannot be improved by any small change of its action probabilities, then it is optimal, provided the gradient is computed with a starting distribution that gives every state some weight. More quantitatively, the gap to the optimal return is at most a constant times a measure of how far the policy is from being stationary. The constant, a *distribution mismatch coefficient*, compares how often the optimal policy visits each state with how often the states are covered during optimization. When some state that matters is rarely visited, the constant is huge and the guarantee weak (Section 14.2).
2. *Plain policy gradient converges, sometimes very slowly.* Softmax policy gradient with exact gradients converges to an optimal policy, and its error decreases like $1/t$ after t iterations. But the constant in front of $1/t$ depends on the problem and on the initial policy, and there are examples where convergence takes a number of iterations exponential in the effective horizon $1/(1-\gamma)$ (Section 14.3).
3. *Natural policy gradient is fast and dimension-free.* The natural gradient of Chapter 12, applied to softmax policies, multiplies each action's probability by the exponential of its advantage. With step size η and $|\mathcal{A}|$ actions per state, its error after T iterations is at most $\frac{\log|\mathcal{A}|}{\eta T} + \frac{1}{(1-\gamma)^2 T}$, whatever the number of states, the initial distribution or the plateaus; with growing step sizes it converges linearly, like policy iteration (Section 14.4).
4. *Entropy regularization buys linear convergence.* Adding an entropy bonus to the rewards,

as SAC does (Section 13.5), makes natural policy gradient converge linearly to the optimal policy of the regularized problem. That policy is slightly suboptimal for the original problem, by at most $\tau \log |\mathcal{A}|/(1-\gamma)$ for temperature τ (the α of SAC), so the temperature trades speed against bias (Section 14.5).

5. *Beyond tables, the guarantees degrade.* With function approximation the same analysis holds up to an approximation error that depends on how well the policy class can represent improvements where the optimal policy goes. Policy gradient also finds the global optimum of linear-quadratic control, although that problem is not convex in the feedback gain. For partially observed problems and for games, no comparable global guarantees exist in general (Section 14.6, Section 14.7).

KEY IDEA

The policy optimization objective is not concave, but in finite MDPs it has no bad local optima: a policy that no small change can improve is optimal. Global convergence of policy gradient follows. How *fast* it converges depends on exploration (how well the states that matter are visited) and on geometry (natural gradients and entropy regularization avoid the plateaus of plain gradients).

In practice, these results explain several things you may have noticed in Chapter 11 and Chapter 12. Why policy gradient learning curves can sit flat for a long time and then rise sharply: a plateau. Why a stochastic initial policy and an entropy bonus help: they keep every action's probability away from zero, where gradients vanish. Why natural gradients, trust regions and PPO's conservative updates work better than plain gradients: they measure steps in the space of policies rather than parameters. And why exploration still matters for "local" methods: the mismatch coefficient is a measure of exploration.

CHECKPOINT

1. Why is the Trail's return in logit space never maximized by any finite (z_0, z_1)?
2. At $z = (-5, -5)$ the Trail's gradient is tiny. Is that point a local maximum?
3. Which quantity in the gradient domination statement measures exploration?

Answers.

1. The optimal policy is deterministic ($p_0 = p_1 = 1$), and a sigmoid equals 1 only in the limit of an infinite logit, so J approaches 3.24 without reaching it.
2. No. The gradient is small but not zero, and following it leads to the global optimum, only slowly. It is a plateau, not a local maximum.
3. The distribution mismatch coefficient, which compares how often the optimal policy visits each state with how often the optimization covers it.

14.2 Gradient domination ●●

We now make the first statement precise. Throughout the chapter the MDP is finite, rewards lie in $[0, 1]$, and $\gamma < 1$, so all values lie in $[0, 1/(1-\gamma)]$. For a distribution ρ over states we write $V^\pi(\rho) = \sum_s \rho(s) V^\pi(s)$, and d^π_ρ for the discounted state-visitation distribution of π started from

ρ (Definition 3.8): $d_\rho^\pi(s) = (1-\gamma)\sum_t \gamma^t \Pr^\pi(s_t = s \mid s_0 \sim \rho)$. Two distributions play different roles. The *performance measure* ρ is where we evaluate the policy. The *optimization measure* μ is the start distribution whose value, $V^\pi(\mu)$, the algorithm ascends. Using a μ different from ρ is a way to explore: it lets the gradient see states that the start distribution reaches rarely.

The direct parameterization. The simplest parameterization takes the action probabilities themselves as parameters: $\theta_{s,a} = \pi(a \mid s)$, constrained to the product of probability simplices. In this parameterization the policy gradient theorem (Theorem 11.1) takes the form

$$\frac{\partial V^\pi(\mu)}{\partial \pi(a \mid s)} = \frac{1}{1-\gamma}\, d_\mu^\pi(s)\, Q^\pi(s,a). \tag{14.1}$$

(The theorem's $\frac{1}{1-\gamma}\mathbb{E}_{s\sim d_\mu^\pi}\big[\sum_a \nabla_\theta \pi(a \mid s)\, Q^\pi(s,a)\big]$ becomes (14.1) because $\nabla_\theta \pi(a \mid s)$ is the unit vector of coordinate (s,a).) In words: raising the probability of a in s increases the return in proportion to how often s is visited and to how good a is there.

Because the parameters are constrained, "stationary" must be defined with care: a policy π is a first-order stationary point if no feasible direction increases $V^\pi(\mu)$ to first order, that is, if $\max_{\bar\pi}(\bar\pi - \pi)^\top \nabla_\pi V^\pi(\mu) = 0$, where the maximum is over all policies $\bar\pi$ and $(\bar\pi - \pi)^\top \nabla_\pi V^\pi(\mu) = \sum_{s,a}(\bar\pi(a \mid s) - \pi(a \mid s))\, \partial V^\pi(\mu)/\partial \pi(a \mid s)$. This maximum is always nonnegative (take $\bar\pi = \pi$), and it measures how far π is from stationarity.

Lemma 14.1 Gradient domination (Agarwal et al., 2021)

For the direct parameterization, every policy π and all state distributions ρ and μ,

$$V^*(\rho) - V^\pi(\rho) \;\le\; \left\|\frac{d_\rho^{\pi^*}}{d_\mu^\pi}\right\|_\infty \max_{\bar\pi} (\bar\pi - \pi)^\top \nabla_\pi V^\pi(\mu) \;\le\; \frac{1}{1-\gamma}\left\|\frac{d_\rho^{\pi^*}}{\mu}\right\|_\infty \max_{\bar\pi} (\bar\pi - \pi)^\top \nabla_\pi V^\pi(\mu),$$

where π^* is an optimal policy, the ratios are taken componentwise over the states where the numerator is positive (and are $+\infty$ if a denominator vanishes there), and absorbing terminal states may be left out.

Proof. By the performance difference lemma (Lemma 3.10),

$$V^*(\rho) - V^\pi(\rho) = \frac{1}{1-\gamma}\sum_s d_\rho^{\pi^*}(s) \sum_a \pi^*(a \mid s)\, A^\pi(s,a) \le \frac{1}{1-\gamma}\sum_s d_\rho^{\pi^*}(s) \max_a A^\pi(s,a).$$

Each $\max_a A^\pi(s,a)$ is nonnegative, because $\sum_a \pi(a \mid s) A^\pi(s,a) = 0$; so we may multiply and divide by $d_\mu^\pi(s)$ and bound the ratio by its maximum:

$$V^*(\rho) - V^\pi(\rho) \le \left\|\frac{d_\rho^{\pi^*}}{d_\mu^\pi}\right\|_\infty \frac{1}{1-\gamma}\sum_s d_\mu^\pi(s) \max_a A^\pi(s,a).$$

The remaining sum is a maximum over policies,

$$\sum_s d_\mu^\pi(s) \max_a A^\pi(s,a) = \max_{\bar\pi} \sum_s d_\mu^\pi(s) \sum_a \bar\pi(a \mid s) A^\pi(s,a),$$

since the maximum can be taken separately in each state. Finally, for every policy $\bar{\pi}$, $\sum_a \bar{\pi}(a \mid s) A^\pi(s,a)$ equals $\sum_a (\bar{\pi}(a \mid s) - \pi(a \mid s)) Q^\pi(s,a)$, because both $\bar{\pi}$ and π sum to one and $\sum_a \pi(a \mid s) Q^\pi(s,a) = V^\pi(s)$. By (14.1) this gives

$$\frac{1}{1-\gamma} \sum_s d_\mu^\pi(s) \sum_a \bar{\pi}(a \mid s) A^\pi(s,a) = (\bar{\pi} - \pi)^\top \nabla_\pi V^\pi(\mu),$$

which proves the first inequality. The second follows from $d_\mu^\pi(s) \geq (1-\gamma)\mu(s)$, which holds because the $t = 0$ term of the visitation sum is $(1-\gamma)\mu(s)$. ■

The lemma says that stationarity implies optimality whenever the mismatch coefficient is finite, for example when $\mu(s) > 0$ in every state: then a policy with $\max_{\bar{\pi}} (\bar{\pi} - \pi)^\top \nabla_\pi V^\pi(\mu) = 0$ has $V^\pi(\rho) = V^*(\rho)$ for every ρ. It also says how much an approximately stationary policy can lose: the stationarity measure times the *distribution mismatch coefficient* $\| d_\rho^{\pi^*} / \mu \|_\infty / (1-\gamma)$. The coefficient is small when μ covers the states that the optimal policy visits, and large when it does not; it is the price of exploration in this theory. The first, finer bound uses the current policy's own visitation d_μ^π, which can be finite even when μ misses states, provided the policy itself goes there.

Example 14.1 Gradient domination on the Trail

Take the uniform policy on the Trail with $\gamma = 0.9$ and $\rho = \mu$ concentrated on Start. The gap is $V^*(\text{Start}) - V^\pi(\text{Start}) = 3.24 - 1.76 = 1.48$. The visitation distribution of the uniform policy is $d_\mu^\pi = (0.1, 0.045, 0.02025)$ on Start, Middle and Goal, and that of the optimal policy is $d_\rho^{\pi^*} = (0.1, 0.09, 0.081)$; their ratios are 1, 2 and 4, so the first mismatch coefficient is 4. By (14.1) the gradient in Start is $(1, 2.52)$ for (exit, go) and in Middle it is $(0.9, 1.62)$. The best direction moves all probability to go in both states, gaining $2.52 - 1.76 = 0.76$ in Start and $1.62 - 1.26 = 0.36$ in Middle, so the stationarity measure is 1.12. The lemma's bound is $4 \times 1.12 = 4.48 \geq 1.48$. The coarser bound is infinite, because μ puts no weight on Middle. With μ uniform over the three nonterminal states instead, both coefficients equal 3, the stationarity measure is 0.64, and the bound is 1.92.

♦

Projected gradient ascent. Gradient domination turns any method that finds approximately stationary points into a method that finds approximately optimal policies. The simplest is projected gradient ascent: take a gradient step on $V^\pi(\mu)$ and project each $\pi(\cdot \mid s)$ back onto the probability simplex. Because $V^\pi(\mu)$ is a smooth function of the direct parameters, with a smoothness constant of order $|\mathcal{A}|/(1-\gamma)^3$, standard nonconvex optimization arguments combined with the lemma give a global rate.

Theorem 14.2 Projected policy gradient (Agarwal et al., 2021; Xiao, 2022)

With the direct parameterization, exact gradients and a constant step size of order $(1-\gamma)^3/(\gamma|\mathcal{A}|)$, projected gradient ascent on $V^\pi(\mu)$ reaches $\min_{t<T} \left(V^*(\rho) - V^{\pi_t}(\rho)\right) \leq \epsilon$ within

$$T = O\Big(\frac{|\mathcal{S}|\,|\mathcal{A}|}{(1-\gamma)^6\,\epsilon^2} \Big\| \frac{d_\rho^{\pi^*}}{\mu} \Big\|_\infty^2 \Big)$$

iterations. Xiao (2022) sharpens the analysis to an $O(1/t)$ rate for the last iterate.

The polynomial dependence on the number of states, the horizon and the mismatch coefficient is typical of the "smooth nonconvex" style of analysis. The rest of the chapter shows how other parameterizations and update rules change these dependences.

14.3 Softmax policy gradient ●●

In practice policies are parameterized by logits. The tabular *softmax* policy has one logit per state-action pair, $\pi_\theta(a \mid s) = e^{\theta_{s,a}} / \sum_b e^{\theta_{s,b}}$, and every positive policy can be represented. By the chain rule, $\partial \pi_\theta(b \mid s)/\partial \theta_{s,a} = \pi_\theta(b \mid s)(\mathbb{1}\{a = b\} - \pi_\theta(a \mid s))$, and (14.1) becomes

$$\frac{\partial V^{\pi_\theta}(\mu)}{\partial \theta_{s,a}} = \frac{1}{1-\gamma}\, d_\mu^{\pi_\theta}(s)\, \pi_\theta(a \mid s)\, A^{\pi_\theta}(s,a). \tag{14.2}$$

Compared with the direct parameterization there is an extra factor $\pi_\theta(a \mid s)$. It is the source of both the good and the bad behavior of softmax policy gradient. It keeps the parameters unconstrained and the policy positive. But it makes the gradient vanish whenever the policy becomes nearly deterministic, including when it becomes nearly deterministic in the *wrong* action, and it makes the optimum lie at infinity, as on the Trail.

Theorem 14.3 Global convergence of softmax policy gradient

Let $\mu(s) > 0$ for every state, and run gradient ascent $\theta_{t+1} = \theta_t + \eta \nabla_\theta V^{\pi_{\theta_t}}(\mu)$ with exact gradients, from any θ_0.

(a) (A. Agarwal et al. 2021) If $\eta \le (1-\gamma)^3/8$, then $V^{\pi_{\theta_t}}(s) \to V^*(s)$ for every state s.

(b) (Mei et al. 2020) If $\eta = (1-\gamma)^3/8$, then $c = \inf_{t \ge 1} \min_s \pi_{\theta_t}(a^*(s) \mid s) > 0$, where $a^*(s)$ is an optimal action, and for every distribution ρ and every $t \ge 1$,

$$V^*(\rho) - V^{\pi_{\theta_t}}(\rho) = O\Big(\frac{|\mathcal{S}|}{c^2 (1-\gamma)^6\, t} \Big\| \frac{d_\mu^{\pi^*}}{\mu} \Big\|_\infty^2 \Big\| \frac{1}{\mu} \Big\|_\infty\Big).$$

Part (a) uses gradient domination in a subtle way: the softmax gradient is the direct gradient multiplied by the action probabilities, so stationarity can be reached by probabilities going to zero rather than by advantages vanishing, and the proof must rule out convergence to such points. Part (b) rests on a *nonuniform Łojasiewicz inequality*: the gradient's norm is at least a constant times $\min_s \pi_\theta(a^*(s) \mid s)$ times the suboptimality, so it is small only if the policy is nearly optimal *or* gives tiny probability to the optimal actions. The constant c controls everything, and it is not under the user's control. The bandit case shows the whole argument in a few lines.

Proposition 14.4 Softmax policy gradient on a bandit

Consider a bandit with K arms and mean rewards $r \in [0,1]^K$; let a^* be a best arm and $r^* = r_{a^*}$. Run softmax policy gradient on $f(\theta) = \pi_\theta^\top r$ with $\theta_0 = 0$ (the uniform policy) and step size

$\eta = 1/3$. Then $\pi_{\theta_t}(a^*)$ never decreases, and for every $t \geq 1$,

$$r^* - \pi_{\theta_t}^\top r \leq \frac{6K^2}{t}.$$

Proof. Write $\pi = \pi_\theta$ and $\bar{r} = \pi^\top r$. The gradient is $u = \nabla f(\theta)$ with components $u_a = \pi_a(r_a - \bar{r})$ (the gradient bandit of Section 2.6).

Smoothness. Differentiating once more, $\nabla^2 f = \operatorname{diag}(u) - \pi u^\top - u\pi^\top$. For any vector y, $|y^\top \nabla^2 f\, y| \leq |\sum_a u_a y_a^2| + 2|\pi^\top y|\,|u^\top y|$. Since $|u_a| \leq 1$, the first term is at most $\|y\|^2$; since $\|\pi\|_2 \leq 1$ and $\|u\|_2 \leq \|u\|_1 = \sum_a \pi_a |r_a - \bar{r}| \leq 1$, the second is at most $2\|y\|^2$. So f is 3-smooth, and the step $\eta = 1/3$ guarantees the ascent property $f(\theta_{t+1}) \geq f(\theta_t) + \frac{1}{6}\|\nabla f(\theta_t)\|^2$.

The best arm's probability never decreases. Suppose $\pi_{a^*} \geq \pi_a$ for all a, which holds at the uniform start. The logit difference $\theta_{a^*} - \theta_a$ changes by $\eta(u_{a^*} - u_a)$. If $r_a \leq \bar{r}$, then $u_a \leq 0 \leq u_{a^*}$. If $r_a > \bar{r}$, then $u_{a^*} = \pi_{a^*}(r^* - \bar{r}) \geq \pi_a(r_a - \bar{r}) = u_a$, because $\pi_{a^*} \geq \pi_a$ and $r^* \geq r_a$. Either way every ratio π_{a^*}/π_a is nondecreasing, so $\pi_{a^*} \geq \pi_a$ persists and $\pi_{a^*} = 1/\sum_a (\pi_a/\pi_{a^*})$ never decreases. In particular $\pi_{\theta_t}(a^*) \geq 1/K$.

A Łojasiewicz inequality. Let $\delta_t = r^* - \pi_{\theta_t}^\top r \geq 0$. The gradient's norm is at least its a^* component: $\|\nabla f(\theta_t)\| \geq u_{a^*} = \pi_{\theta_t}(a^*)\,\delta_t \geq \delta_t/K$.

The rate. Combining, $\delta_{t+1} \leq \delta_t - \frac{1}{6K^2}\delta_t^2 = \delta_t(1 - a\delta_t)$ with $a = 1/(6K^2)$ and $a\delta_t < 1$. If $\delta_{t+1} > 0$, then $\frac{1}{\delta_{t+1}} \geq \frac{1}{\delta_t(1-a\delta_t)} \geq \frac{1+a\delta_t}{\delta_t} = \frac{1}{\delta_t} + a$, using $1/(1-x) \geq 1 + x$. Hence $1/\delta_t \geq ta$, that is, $\delta_t \leq 6K^2/t$. ∎

The proof shows where the constant comes from: the bound $\pi_{\theta_t}(a^*) \geq 1/K$ in the Łojasiewicz step. From a general initialization the best arm's probability need not be monotone, but the same argument with $c = \inf_t \pi_{\theta_t}(a^*)$ in place of $1/K$ gives the rate $6/(c^2 t)$. For two arms $\pi_{\theta_t}(a^*)$ never decreases (Exercise 14.5), so $c = \pi_{\theta_0}(a^*)$, which is tiny when the initial logits strongly prefer the wrong arm. For two arms with rewards 1 and 0.5, starting from $\theta_0 = (-10, 0)$, the best arm's initial probability is 4.5×10^{-5} and the initial gradient is 2.3×10^{-5}; it takes 66,144 iterations before the best arm becomes the more likely one (Exercise 14.5). The rate $O(1/t)$ itself cannot be improved: Mei et al. (2020) show a matching $\Omega(1/t)$ lower bound for softmax policy gradient.

Exponential slowness. In MDPs the trouble compounds over states. Li et al. (2023) construct MDPs in which softmax policy gradient, with exact gradients, a benign initial policy and an initial distribution whose mismatch coefficient is moderate, needs a number of iterations exponential in the effective horizon $1/(1-\gamma)$; their lower bound, of order $|\mathcal{S}|^{2^{\Omega(1/(1-\gamma))}}/\eta$ for step size η, is in fact doubly exponential. The mechanism is visible in a simpler chain (Figure 14.2). In each of ten states the agent can quit, ending the episode with reward 0.1, or move right; only the last right earns 1. With $\gamma = 0.9$ the optimal policy goes right throughout and earns $0.9^9 = 0.387$ from the first state. From the uniform policy, the reward at the end is reached with probability 2^{-10}, so in the early states quitting looks better than moving on, and their policies learn to quit. That starves the later states of visits, and by (14.2) of gradient, so they never learn that moving on pays. When the gradient is computed with the start-state distribution, softmax policy gradient with $\eta = 1$ is still stuck after 50,000 iterations at the policy that quits at once, with a gap of 0.287.

At iteration 3,000 the stationarity measure of Lemma 14.1 is only 2.5×10^{-4}, but the mismatch coefficient is about 45,000: exactly the situation the lemma warns about. With μ uniform over the chain the same method reaches a gap below 0.01 in about 1,600 iterations; natural policy gradient takes five.

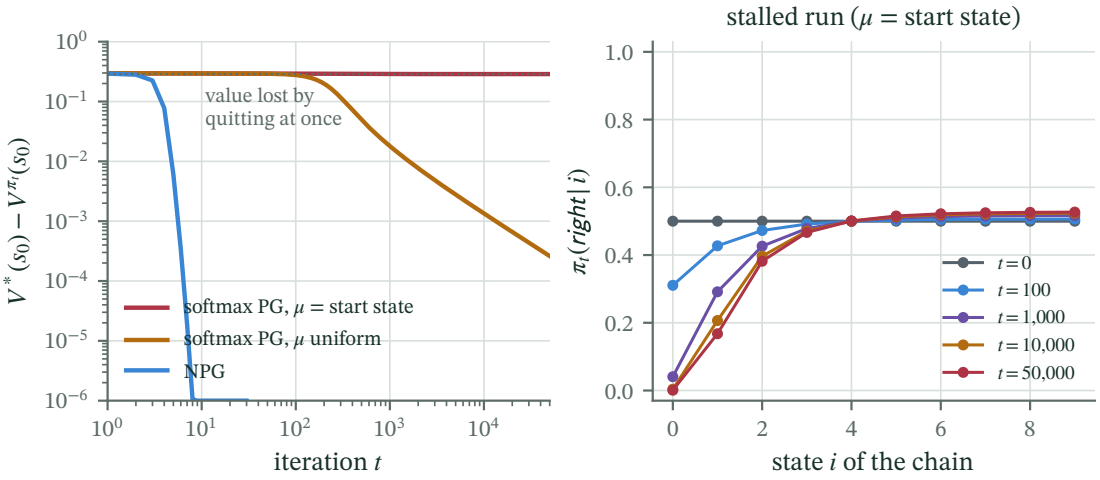


Figure 14.2. A chain where softmax policy gradient stalls (10 states, $\gamma = 0.9$, $\eta = 1$, exact gradients). Left: suboptimality of the start state. With the start-state distribution as μ (rose), the policy gets stuck near "quit at once" (dotted line: the value lost by quitting, 0.287); with μ uniform over the chain (amber), it converges after a long plateau; natural policy gradient (blue) converges in a few iterations. Right: the probability of right in each state during the stalled run. The first states learn to quit, which starves the later states of visits; they stay near $\frac{1}{2}$.

Three remedies emerge from the analysis, and the next two sections develop two of them. A better geometry: natural policy gradient's update does not contain the visitation factor at all (Section 14.4). Regularization: an entropy bonus keeps probabilities away from zero (Section 14.5). Normalization: dividing the gradient by its norm also escapes plateaus, and Mei et al. (2021) show that, with exact gradients, a normalized variant of softmax policy gradient converges linearly.

14.4 Natural policy gradient and policy mirror descent ●●

The natural policy gradient (Kakade 2001) preconditions the gradient with the inverse of the Fisher information matrix (Section 12.4, (12.3)): $\theta_{t+1} = \theta_t + \eta\, F_\rho(\theta_t)^\dagger \nabla_\theta V^{\pi_{\theta_t}}(\rho)$, where $F_\rho(\theta) = \mathbb{E}_{s\sim d_\rho^{\pi_\theta}} \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)^\top\big]$ and $\dagger$ is the Moore–Penrose pseudo-inverse. For tabular softmax policies the update has a closed form.

Lemma 14.5 Natural policy gradient is multiplicative weights

For the tabular softmax parameterization, if $d_\rho^{\pi_\theta}(s) > 0$ for every state, then $F_\rho(\theta)^\dagger \nabla_\theta V^{\pi_\theta}(\rho)$ equals $\frac{1}{1-\gamma} A^{\pi_\theta}$ minus, in each state s, the average of $\frac{1}{1-\gamma} A^{\pi_\theta}(s, \cdot)$ over actions. Consequently the

natural policy gradient step is

$$\pi_{t+1}(a \mid s) = \frac{\pi_t(a \mid s)\exp\big(\eta A^{\pi_t}(s,a)/(1-\gamma)\big)}{Z_t(s)}, \qquad Z_t(s) = \sum_b \pi_t(b \mid s)\exp\big(\eta A^{\pi_t}(s,b)/(1-\gamma)\big). \tag{14.3}$$

Proof. The score is $\nabla_\theta \log \pi_\theta(a \mid s) = e_{s,a} - \sum_b \pi_\theta(b \mid s)\, e_{s,b}$, where $e_{s,a}$ is a unit vector; it is supported on the block of coordinates of state s. Let $w = A^{\pi_\theta}/(1-\gamma)$, viewed as a vector indexed by (s,a). Then $\nabla_\theta \log \pi_\theta(a \mid s)^\top w = w_{s,a} - \sum_b \pi_\theta(b \mid s) w_{s,b} = A^{\pi_\theta}(s,a)/(1-\gamma)$, because the advantages average to zero under π_θ. Hence

$$F_\rho(\theta)\, w = \mathbb{E}_{s\sim d_\rho^{\pi_\theta}} \mathbb{E}_{a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\, \frac{A^{\pi_\theta}(s,a)}{1-\gamma}\Big] = \nabla_\theta V^{\pi_\theta}(\rho),$$

by the policy gradient theorem with the advantage in place of Q (the baseline $V^{\pi_\theta}(s)$ does not change the expectation, Section 11.5). So w solves $Fx = \nabla V$. The Fisher matrix is block diagonal, with block $d_\rho^{\pi_\theta}(s)\big(\mathrm{diag}(\pi_s) - \pi_s \pi_s^\top\big)$ for state s, where $\pi_s = \pi_\theta(\cdot \mid s)$; for a vector v supported on that block, $v^\top F v = d_\rho^{\pi_\theta}(s)\,\mathrm{Var}_{a\sim\pi_s}(v_a)$, which vanishes exactly when v is constant on the block. So the null space of F consists of the vectors that are constant within each state. The pseudo-inverse returns the solution of $Fx = \nabla V$ that is orthogonal to the null space, which is w with its mean in each state removed. Constants added to all logits of a state do not change the softmax policy, so the step $\theta + \eta F^\dagger \nabla V$ produces (14.3). ∎

The update (14.3) is the *multiplicative weights* (or *Hedge*) algorithm of online learning, run in every state with the advantages as payoffs. Two features distinguish it from softmax policy gradient (14.2). The visitation $d_\rho^{\pi_t}(s)$ has disappeared, so rarely visited states are updated as fast as frequent ones; this is why NPG solves the chain of Figure 14.2 in five iterations. And the factor $\pi_t(a \mid s)$ has disappeared, so a nearly deterministic but wrong choice can be corrected quickly. Since $V^{\pi_t}(s)$ does not depend on a, we may write Q^{π_t} in place of A^{π_t} in (14.3). As $\eta \to \infty$ the update puts all probability on the greedy actions: NPG with infinite step size is policy iteration (Algorithm 4.2).

Theorem 14.6 Rate of natural policy gradient (Agarwal et al., 2021)

Let rewards lie in $[0,1]$, let π_0 be the uniform policy, and run the update (14.3) with any step size $\eta > 0$. Then $V^{\pi_t}(s)$ is nondecreasing in t for every state s, and for every distribution ρ and every $T \geq 1$,

$$V^*(\rho) - V^{\pi_T}(\rho) \leq \frac{\log|\mathcal{A}|}{\eta\, T} + \frac{1}{(1-\gamma)^2\, T}.$$

The bound depends neither on the number of states nor on any distribution mismatch coefficient, and it holds for every ρ at once.

Proof. *Step 1: improvement.* By Jensen's inequality (the logarithm is concave) and $\sum_a \pi_t(a \mid s) A^{\pi_t}(s,a) = 0$,

$$\log Z_t(s) \geq \sum_a \pi_t(a \mid s)\, \frac{\eta A^{\pi_t}(s,a)}{1-\gamma} = 0.$$

Taking logarithms in (14.3) gives $A^{\pi_t}(s,a) = \frac{1-\gamma}{\eta}\log\frac{\pi_{t+1}(a|s)Z_t(s)}{\pi_t(a|s)}$, and hence

$$\sum_a \pi_{t+1}(a \mid s)A^{\pi_t}(s,a) = \frac{1-\gamma}{\eta}\Big[D_{\mathrm{KL}}\big(\pi_{t+1}(\cdot \mid s)\,\|\,\pi_t(\cdot \mid s)\big) + \log Z_t(s)\Big] \geq \frac{1-\gamma}{\eta}\log Z_t(s) \geq 0.$$

For any start distribution ν, the performance difference lemma then gives

$$\begin{aligned} V^{\pi_{t+1}}(\nu) - V^{\pi_t}(\nu) &= \frac{1}{1-\gamma}\sum_s d_\nu^{\pi_{t+1}}(s)\sum_a \pi_{t+1}(a \mid s)A^{\pi_t}(s,a) \\ &\geq \frac{1}{\eta}\sum_s d_\nu^{\pi_{t+1}}(s)\log Z_t(s) \;\geq\; \frac{1-\gamma}{\eta}\sum_s \nu(s)\log Z_t(s), \end{aligned}$$

using $d_\nu^{\pi_{t+1}} \geq (1-\gamma)\nu$ and $\log Z_t \geq 0$. With ν concentrated on one state, this shows that every $V^{\pi_t}(s)$ is nondecreasing.

Step 2: a telescoping comparison with π^*. Let $d^* = d_\rho^{\pi^*}$. By the performance difference lemma and the expression for A^{π_t} from Step 1,

$$\begin{aligned} V^*(\rho) - V^{\pi_t}(\rho) &= \frac{1}{1-\gamma}\sum_s d^*(s)\sum_a \pi^*(a \mid s)A^{\pi_t}(s,a) \\ &= \frac{1}{\eta}\sum_s d^*(s)\sum_a \pi^*(a \mid s)\log\frac{\pi_{t+1}(a \mid s)Z_t(s)}{\pi_t(a \mid s)} \\ &= \frac{1}{\eta}\sum_s d^*(s)\Big[D_{\mathrm{KL}}\big(\pi_s^* \,\|\, \pi_{t,s}\big) - D_{\mathrm{KL}}\big(\pi_s^* \,\|\, \pi_{t+1,s}\big) + \log Z_t(s)\Big], \end{aligned}$$

where $\pi_{t,s} = \pi_t(\cdot \mid s)$. Step 1 with $\nu = d^*$ bounds the last term: $\frac{1}{\eta}\sum_s d^*(s)\log Z_t(s) \leq \frac{1}{1-\gamma}\big(V^{\pi_{t+1}}(d^*) - V^{\pi_t}(d^*)\big)$. Summing over $t = 0, \dots, T-1$, the KL terms and the value terms telescope:

$$\sum_{t=0}^{T-1}\big(V^*(\rho) - V^{\pi_t}(\rho)\big) \leq \frac{1}{\eta}\sum_s d^*(s)D_{\mathrm{KL}}\big(\pi_s^* \,\|\, \pi_{0,s}\big) + \frac{V^{\pi_T}(d^*) - V^{\pi_0}(d^*)}{1-\gamma} \leq \frac{\log|\mathcal{A}|}{\eta} + \frac{1}{(1-\gamma)^2}.$$

Here we dropped $-D_{\mathrm{KL}}(\pi_s^* \| \pi_{T,s}) \leq 0$, used $D_{\mathrm{KL}}(p \,\|\, \text{uniform}) \leq \log|\mathcal{A}|$, and used that values lie in $[0, 1/(1-\gamma)]$. Finally, by Step 1, $V^{\pi_T}(\rho) \geq V^{\pi_t}(\rho)$ for every $t \leq T$, so $V^*(\rho) - V^{\pi_T}(\rho)$ is at most the average of the T terms on the left, which proves the theorem. ■

The proof is the classical regret analysis of multiplicative weights, run in every state and weighted by the optimal policy's visitation, plus one twist: the term $\log Z_t$, which in online learning is controlled by bounding the payoffs, is controlled here by the policy's own improvement. That is why the bound does not degrade as η grows; with $\eta \to \infty$ it becomes $\frac{1}{(1-\gamma)^2 T}$, a rate for policy iteration.

Figure 14.3 compares the methods on a random MDP with 10 states, 4 actions, $\gamma = 0.9$ and rewards in $[0,1]$. NPG with $\eta = 0.1$ has gap 0.27 after 10 iterations, 1.3×10^{-3} after 100 and 4.3×10^{-5} after 1,000, while the bound of Theorem 14.6 is 11.4, 1.14 and 0.114: valid, but loose by orders of magnitude, as worst-case bounds usually are. Softmax policy gradient with the much larger step size $\eta = 1$ is slower: its gaps are 1.64, 0.10 and 0.008, decreasing like $1/t$.

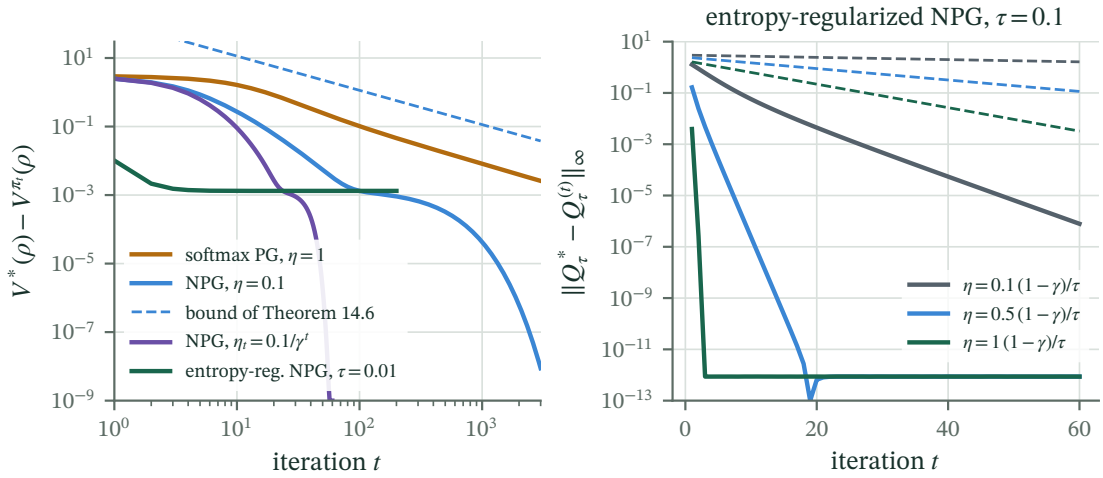


Figure 14.3. Suboptimality against iterations on a random MDP (10 states, 4 actions, $\gamma = 0.9$, rewards in $[0, 1]$, ρ uniform, exact gradients). Left: softmax policy gradient decreases like $1/t$; NPG is much faster and stays below the bound of Theorem 14.6 (dashed); NPG with geometrically growing step sizes converges linearly; entropy-regularized NPG ($\tau = 0.01$) converges within a few iterations to its regularized optimum and then stays at its bias, 0.0013. Right: entropy-regularized NPG with $\tau = 0.1$ and three step sizes; the distance to the regularized optimal Q decreases geometrically and stays below the bounds of Theorem 14.7 (dashed lines of the same colors).

Linear convergence with growing step sizes. The $O(1/T)$ rate is not the end of the story. Because larger steps never hurt in the analysis, one can let them grow. Xiao (2022) and Lan (2023) show that with step sizes that increase geometrically, for example $\eta_{t+1} = \eta_t/\gamma$, NPG and the more general policy mirror descent methods below converge *linearly*, without any regularization. Johnson et al. (2023) show that with suitably adaptive step sizes they attain the rate γ^t of policy iteration, and that this rate is optimal for this family of methods. In Figure 14.3, NPG with $\eta_t = 0.1/\gamma^t$ reaches a gap of 9×10^{-4} after 30 iterations.

14.4.1 Policy mirror descent

The update (14.3) is one instance of a general template. *Policy mirror descent* (PMD) updates the policy in every state by

$$\pi_{t+1}(\cdot \mid s) = \arg\max_{p \in \Delta(\mathcal{A})} \Big\{ \eta_t \sum_a p(a)\, Q^{\pi_t}(s,a) - D\big(p, \pi_t(\cdot \mid s)\big) \Big\},$$

where D is a Bregman divergence that keeps the new policy close to the old one: a linearized improvement step with a proximal penalty. With $D = D_{\mathrm{KL}}$ the solution is $\pi_{t+1} \propto \pi_t e^{\eta_t Q^{\pi_t}}$, which is NPG with $\eta_t = \eta/(1-\gamma)$. With the squared Euclidean distance it is the projected Q-ascent $\pi_{t+1}(\cdot \mid s) = \mathrm{Proj}_\Delta\big(\pi_t(\cdot \mid s) + \eta_t Q^{\pi_t}(s,\cdot)\big)$ (Exercise 14.12). The analyses of Xiao (2022) and Lan (2023) cover both, and Lan (2023) also treats regularized problems and sampled Q-functions.

The template explains the trust-region methods of Chapter 12. TRPO (Algorithm 12.1) maximizes the surrogate objective (12.1), which is the linearization of the return around the old policy that appears in PMD, subject to a bound on the average KL divergence: a constrained form

of the KL-proximal step. PPO's clipped objective (12.6) is a cruder way of keeping the new policy near the old one. Both solve the per-state proximal problem only approximately: with a parameterized policy shared across states, with sampled advantage estimates, and with a few epochs of stochastic gradient steps. Shani et al. (2020) analyze an exact, adaptive version of TRPO as mirror descent and prove global convergence rates, which are faster for regularized MDPs, and Tomar et al. (2022) build a practical algorithm directly on mirror descent. How much of the theory survives PPO's approximations is an open question (Exercise 14.14).

14.5 Entropy regularization ●●

Entropy regularization, introduced for SAC in Section 13.5, also has a clean theory. For a temperature $\tau > 0$ (called α in Section 13.5; we follow the literature on regularized MDPs and write τ), the *regularized value* of a policy is

$$V_\tau^\pi(s) = \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t \big(r(s_t, a_t) - \tau \log \pi(a_t \mid s_t)\big) \mid s_0 = s\Big],$$

the expected return with the entropy bonus of Section 13.5, and $Q_\tau^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}[V_\tau^\pi(s')]$ satisfies the soft Bellman equation (13.6). The regularized problem has a unique optimal policy $\pi_\tau^*(a \mid s) \propto \exp(Q_\tau^*(s,a)/\tau)$, which is stochastic. Geist et al. (2019) develop the theory of such *regularized MDPs* for general convex regularizers, of which the negative entropy is the most important.

Regularization makes the optimization problem better conditioned: the objective now has a unique maximizer, attained at a finite point in logit space, and the entropy term curves the landscape near it. The natural policy gradient of the regularized objective has a closed form like (14.3):

$$\pi_{t+1}(a \mid s) \propto \pi_t(a \mid s)^{1 - \frac{\eta\tau}{1-\gamma}} \exp\Big(\frac{\eta\, Q_\tau^{\pi_t}(s,a)}{1-\gamma}\Big), \tag{14.4}$$

a geometric average of the old policy and the Boltzmann policy of the current soft Q-function. For $\eta = (1-\gamma)/\tau$, the largest step allowed below, the old policy drops out and the update becomes soft policy iteration, $\pi_{t+1} \propto \exp(Q_\tau^{\pi_t}/\tau)$.

Theorem 14.7 Linear convergence of entropy-regularized NPG (Cen et al., 2022)

Let rewards lie in $[0,1]$, $\tau > 0$ and $0 < \eta \le (1-\gamma)/\tau$, and run (14.4) from a policy π_0 with full support. Then for every $t \ge 0$,

$$\big\|Q_\tau^* - Q_\tau^{\pi_{t+1}}\big\|_\infty \le C_1\, \gamma\, (1-\eta\tau)^t, \qquad \big\|\log \pi_\tau^* - \log \pi_{t+1}\big\|_\infty \le \frac{2C_1}{\tau}(1-\eta\tau)^t,$$

where $C_1 = \|Q_\tau^* - Q_\tau^{\pi_0}\|_\infty + 2\tau\big(1 - \frac{\eta\tau}{1-\gamma}\big)\|\log \pi_\tau^* - \log \pi_0\|_\infty$.

The rate $1-\eta\tau$ lies between γ (for the largest step, soft policy iteration) and 1, and, like the bound of Theorem 14.6, it does not depend on the number of states or on mismatch coefficients. For soft policy iteration the result reduces to $\|Q_\tau^* - Q_\tau^{\pi_t}\|_\infty \le \gamma^t \|Q_\tau^* - Q_\tau^{\pi_0}\|_\infty$, which follows from a short

contraction argument (Exercise 14.9); the general case of Cen et al. (2022) needs a more careful analysis of the interpolation between the old policy and the Boltzmann policy. For the plain (not natural) softmax policy gradient, Mei et al. (2020) prove that entropy regularization also yields linear convergence, but with constants that depend on the problem, as in Theorem 14.3. The right panel of Figure 14.3 shows the theorem at work with $\tau = 0.1$: with $\eta = 0.1(1-\gamma)/\tau$ the error falls from 1.8 to 4×10^{-3} in 20 iterations, and with the two larger steps it reaches machine precision within about 20 and 5 iterations; every curve stays below its bound.

The speed has a price: the regularized optimum is not the optimal policy. The price is small and explicit.

Proposition 14.8 The bias of entropy regularization

For every distribution ρ, $0 \le V^*(\rho) - V^{\pi_\tau^*}(\rho) \le \dfrac{\tau \log |\mathcal{A}|}{1-\gamma}$.

Proof. The entropy of any distribution over $\mathcal{A}$ lies between 0 and $\log|\mathcal{A}|$, so for every policy π the regularized value differs from the ordinary one by a discounted sum of terms between 0 and $\tau \log|\mathcal{A}|$: $V^\pi(\rho) \le V_\tau^\pi(\rho) \le V^\pi(\rho) + \frac{\tau \log|\mathcal{A}|}{1-\gamma}$. Applying the right inequality to π_τ^*, the optimality of π_τ^* for the regularized problem, and the left inequality to π^*,

$$V^{\pi_\tau^*}(\rho) \ge V_\tau^{\pi_\tau^*}(\rho) - \frac{\tau \log|\mathcal{A}|}{1-\gamma} \ge V_\tau^{\pi^*}(\rho) - \frac{\tau \log|\mathcal{A}|}{1-\gamma} \ge V^*(\rho) - \frac{\tau \log|\mathcal{A}|}{1-\gamma}.$$

The left inequality of the proposition holds because π^* is optimal. ■

On the random MDP of Figure 14.3 the actual bias is much smaller than the bound: 0.0013 for $\tau = 0.01$ (bound 0.139) and 0.27 for $\tau = 0.1$ (bound 1.39). On the soft Trail of Section 13.5, with temperature 1, it is 0.41 against a bound of 6.93. The bound is nonetheless useful because it is uniform: it tells us that a temperature of order $(1-\gamma)\epsilon/\log|\mathcal{A}|$ is small enough for an ϵ-optimal policy, whatever the MDP. Combining Proposition 14.8 with Theorem 14.7 gives a method that converges linearly to an ϵ-optimal policy; the temperature trades the bias (proportional to τ) against the conditioning that makes convergence fast. In practice the temperature is often adapted during training rather than fixed; SAC's automatic tuning (Section 13.6.2) adjusts it to keep the policy's entropy near a target. The same KL- or entropy-regularized objectives appear in the fine-tuning of language models, where the KL penalty to a reference policy plays the role of the regularizer (Section 22.2, (22.2)).

LAB

Lab 14.1: Why policy gradients find optimal policies, and how fast (`labs/ch14_pg_theory.py`, about 40 minutes; the script runs in seconds). Run `python3 labs/ch14_pg_theory.py`. You will check the non-concavity and the plateaus of the Trail in logit space, verify Lemma 14.1 on the Trail and on random MDPs, compare softmax policy gradient with natural policy gradient and with the bound of Theorem 14.6, reproduce the stall of Figure 14.2, and measure the speed and the bias of entropy-regularized NPG. All computations are exact, with `rlcourse.pg_theory`.

14.6 Function approximation ●●●

The tabular results assume a parameter for every state-action pair and exact gradients. Real policies share parameters across states, and gradients are estimated from samples. What survives?

Log-linear policies and compatible approximation. Consider policies $\pi_\theta(a \mid s) \propto \exp(\theta^\top \phi(s,a))$ with features $\phi(s,a) \in \mathbb{R}^d$. Their score is $\nabla_\theta \log \pi_\theta(a \mid s) = \phi(s,a) - \mathbb{E}_{b\sim\pi_\theta}[\phi(s,b)]$, and the natural gradient direction is the solution $w^\star$ of a regression: fit the advantages with the scores as features, $w^\star \in \arg\min_w \mathbb{E}_{s\sim d_\rho^{\pi_\theta},\, a\sim\pi_\theta}\big[(w^\top \nabla_\theta \log \pi_\theta(a \mid s) - A^{\pi_\theta}(s,a))^2\big]$. This is the compatible function approximation of Sutton et al. (2000) and Kakade (2001): a critic that is linear in the score features is exactly what the natural gradient needs. The NPG update $\theta \leftarrow \theta + \eta w^\star$ multiplies the policy by $\exp(\eta\, w^{\star\top}\phi(s,a))$ up to normalization, a multiplicative-weights update with the *fitted* advantages. A variant, Q-NPG, fits Q^{π_θ} on the features ϕ instead.

Approximation and transfer error. A. Agarwal et al. (2021) extend the proof of Theorem 14.6 to this setting. Their bound on the suboptimality after T iterations has three parts: an optimization term of order $\frac{1}{1-\gamma}\sqrt{\log|\mathcal{A}|/T}$ (the step size must now be tuned, as in online learning with approximate payoffs); a statistical term from estimating each regression with finitely many samples; and an approximation term that does not vanish with T. The approximation term is governed by the *transfer error*: the error of the fitted regression measured not where it was fitted, under d^{π_t}, but under the state distribution of the optimal policy, the comparator in the telescoping argument. The two are related by a *relative condition number* of the feature covariance under the two distributions, which plays the role of the mismatch coefficient. An unfavorable case is easy to picture: if the current policy never visits the states where the optimal policy goes, a critic can fit perfectly where the data are and be wrong exactly where it matters (Exercise 14.13).

Sampled gradients and actor-critic methods. With stochastic gradients, policy gradient methods converge to approximately stationary points with polynomially many samples (Yuan et al. 2022), and gradient domination converts stationarity into optimality in the tabular case. For actor-critic methods, Konda and Tsitsiklis (2000) proved asymptotic convergence of two-timescale algorithms with linear critics on the compatible features: the critic learns faster than the actor, and the actor's gradients become asymptotically small (exactly so for a $\lambda = 1$ critic). Finite-time analyses of two-timescale and natural actor-critic methods give polynomial sample complexities for reaching stationary points and, for natural actor-critic, near-optimal policies up to the approximation error (Wu et al. 2020; Xu et al. 2020). For overparameterized neural policies and critics trained in a regime close to their linearization, neural versions of PPO, TRPO and policy gradient have been shown to converge to globally optimal policies at sublinear rates, up to approximation errors (Liu et al. 2019; Wang et al. 2020).

Exploration. All of these guarantees assume that the optimization measure, or the policies themselves, cover the states that matter; the mismatch coefficients and condition numbers measure how far that holds. In problems that require deliberate exploration they can be exponentially large, and local policy search alone is not enough. Optimistic policy optimization adds exploration bonuses to the evaluated Q-functions and achieves $\sqrt{T}$-type regret in episodic MDPs whose transition kernel is linear in known features (Cai et al. 2020), and PC-PG builds a cover of policies that explore different parts of the state space and starts policy gradient from it (Agarwal,

Henaff, et al. 2020). These methods bring the optimism of Section 16.2 into policy optimization.

14.7 Beyond tabular MDPs ●●●

Linear-quadratic control. The linear-quadratic regulator (LQR, Section 4.10.1) has linear dynamics $x_{t+1} = Ax_t + Bu_t$, quadratic costs $x_t^\top Q x_t + u_t^\top R u_t$ with positive definite Q and R, and random initial states $x_0 \sim \mathcal{D}$. Its optimal policy is a linear feedback $u_t = -K^* x_t$, computable from the Riccati equation. Policy gradient searches directly over the gain matrix K to minimize $C(K) = \mathbb{E}_{x_0}\big[\sum_t (x_t^\top Q x_t + u_t^\top R u_t)\big]$ under $u_t = -Kx_t$. The function C is not convex in K: the set of stabilizing gains, where C is finite, is itself not convex in general (Exercise 14.11 gives a two-dimensional example). Yet Fazel et al. (2018) proved that policy gradient converges globally. The key facts mirror the tabular theory. The gradient has the closed form $\nabla C(K) = 2\big((R+B^\top P_K B)K - B^\top P_K A\big)\Sigma_K$, where P_K solves the Lyapunov equation $P_K = Q + K^\top R K + (A-BK)^\top P_K (A-BK)$ and $\Sigma_K = \mathbb{E}\big[\sum_t x_t x_t^\top\big]$ is the state covariance accumulated along trajectories; and the cost satisfies a gradient domination inequality $C(K) - C(K^*) \le \kappa \|\nabla C(K)\|_F^2$, with κ depending on Σ_{K^*}, on the smallest eigenvalue of $\mathbb{E}[x_0 x_0^\top]$ and on that of R. With suitable step sizes, gradient descent, natural gradient descent ($K \leftarrow K - \eta \nabla C(K)\Sigma_K^{-1}$) and a Gauss–Newton variant all converge linearly from any stabilizing gain, and versions with sampled, zeroth-order gradient estimates need polynomially many samples. Hu et al. (2023) survey the many extensions to robust, risk-sensitive, output-feedback and other control problems.

Partial observability. When the agent sees only observations (Chapter 19), a policy that maps observations to actions is still a parameterized policy, and the policy gradient theorem still gives unbiased gradients. But the landscape changes. The best memoryless policy may need to be stochastic, gradient domination fails, and the objective can have genuine local optima. Finding an optimal memoryless stochastic policy is computationally hard in general (Vlassis et al. 2012). With memory (recurrent policies), the class is richer but the optimization is no easier. Policy gradient remains a practical tool for partially observed problems, with only local guarantees.

Several agents. When several agents run policy gradient at the same time (Chapter 21), each faces a nonstationary problem, and the joint dynamics need not converge. In two-player zero-sum games, simultaneous gradient play can cycle around an equilibrium instead of converging to it (Chapter 21 discusses the matching-pennies example). Some structure restores guarantees: with two-timescale step sizes, independent policy gradient converges to a Nash equilibrium in two-player zero-sum Markov games (Daskalakis et al. 2020), and in Markov potential games, where all agents' incentives derive from one potential function, independent policy gradient converges to Nash equilibria (Leonardos et al. 2022). For general-sum games, convergence to Nash equilibria cannot be expected from such simple dynamics.

RESEARCH NOTE

Open problems: (1) A theory of PPO as it is used, with clipping, minibatches, shared networks and advantage normalization, rather than of exact mirror descent. (2) Landscapes of policy optimization with neural networks outside the linearized regime: when do the tabular conclusions (no spurious optima, plateaus caused by mismatch) carry over? (3) Sample-efficient exploration for policy-based methods in rich observation spaces, and the right analogue of the mismatch coefficient there. (4) Optimal step-size schedules and regularization for policy optimization with function approximation. (5) The theory of KL-regularized policy optimization for language models, where the action space is enormous and rewards come from learned models (Chapter 22).

14.8 Summary

- The expected return is not a concave function of the policy parameters. On the Trail in logit space it has plateaus, and its maximum lies at infinity. But in finite MDPs it has no spurious local optima.
- (●●) The gradient domination lemma bounds the suboptimality of any policy by a distribution mismatch coefficient times a measure of stationarity. Stationary points are optimal when the optimization distribution covers all states, and the mismatch coefficient measures the cost of poor coverage.
- (●●) Softmax policy gradient with exact gradients converges globally, at rate $O(1/t)$ with constants that depend on the problem and the initialization; for bandits the rate can be proved in a few lines. The constants can be exponentially bad, and on a simple chain the method stalls for tens of thousands of iterations when its distribution does not cover the chain.
- (●●) Natural policy gradient for softmax policies is a multiplicative-weights update with advantages as payoffs. It improves monotonically and converges at the dimension-free rate $\frac{\log|\mathcal{A}|}{\eta T} + \frac{1}{(1-\gamma)^2 T}$, and linearly with growing step sizes. It is the KL instance of policy mirror descent, of which TRPO and PPO are approximations.
- (●●) Entropy-regularized natural policy gradient converges linearly, at rate $1 - \eta\tau$, to the regularized optimum, whose suboptimality is at most $\tau \log|\mathcal{A}|/(1-\gamma)$.
- (●●●) With function approximation the guarantees hold up to transfer and statistical errors; linear-quadratic control enjoys global convergence despite non-convexity; partially observed problems and games lack comparable guarantees in general.

The next part of the book turns from optimizing policies to the questions that decide whether reinforcement learning works in practice: learning models, exploring, and learning from fixed data.

14.9 Further Reading

The central reference is A. Agarwal et al. (2021), which proves gradient domination, the convergence of projected and softmax policy gradient, the rate of natural policy gradient and its extension to function approximation, with clear proofs; the book draft of Agarwal et al. (2022) presents the same material as lecture notes. Mei et al. (2020) give the $O(1/t)$ rate of softmax

policy gradient, the nonuniform Łojasiewicz inequality and the effect of entropy regularization, and Li et al. (2023) show how slow softmax policy gradient can be. Bhandari and Russo (2024) study when policy gradient finds global optima in more general settings, including problems with structure beyond finite MDPs, and Bhandari and Russo (2021) show that policy gradient methods with large steps converge linearly, as policy iteration does.

For the natural gradient and mirror-descent view, start with Kakade (2001) and the conservative policy iteration of Kakade and Langford (2002), then read Xiao (2022) and Lan (2023) for linear convergence of policy mirror descent, and Johnson et al. (2023) for the optimal rate. Regularized MDPs are developed by Geist et al. (2019), and the linear convergence of entropy-regularized natural policy gradient by Cen et al. (2022). Shani et al. (2020) analyze TRPO as mirror descent.

(●●●) For function approximation, actor-critic analyses and exploration, see Konda and Tsitsiklis (2000), Wu et al. (2020), Xu et al. (2020), Liu et al. (2019), Cai et al. (2020) and Agarwal, Henaff, et al. (2020). For linear-quadratic control, read Fazel et al. (2018) and the survey of Hu et al. (2023). For games, Daskalakis et al. (2020) and Leonardos et al. (2022); for partial observability, Vlassis et al. (2012).

14.10 Exercises

Exercise 14.1 ● The Trail in logit space

With $\gamma = 0.9$, the Trail's return in logit space is $J(z_0, z_1) = 1+\sigma(z_0)(0.8+1.44\,\sigma(z_1))$. (a) Compute J and its gradient at $z = (0,0)$, $(-5,-5)$ and $(5,5)$. (b) Using the points $z_0 = -4, -2, 0$ on the line $z_1 = 0$, show that J is not concave. (c) Show that the gradient never vanishes, and explain why gradient ascent nevertheless never reaches a maximizer. What happens to the gradient as $z_0, z_1 \to \infty$?

Exercise 14.2 ● The direct parameterization of the Trail

In the probability parameters, $J(p_0, p_1) = 1+0.8p_0+1.44p_0p_1$ on the square $[0,1]^2$. (a) Compute the Hessian of J and show that J is neither concave nor convex. (b) Find J along the diagonal $p_0 = p_1 = t$. (c) Find every point of the square at which no feasible direction increases J to first order, and conclude that J has no spurious local maxima.

Exercise 14.3 ●● Policy gradients in two parameterizations

(a) Derive (14.1) from the policy gradient theorem, and (14.2) from (14.1) by the chain rule. (b) For the uniform Trail policy with $\gamma = 0.9$ and μ concentrated on Start, compute the direct and the softmax gradients for both actions in Start and Middle. (c) Check that the softmax gradients agree with the logit gradient of Exercise 14.1 at $z = (0,0)$.

Exercise 14.4 ●● Distribution mismatch

(a) Reproduce the numbers of Example 14.1: the visitation distributions, the two mismatch coefficients for μ concentrated on Start and for μ uniform over the nonterminal states, and the stationarity measures. (b) Explain why the finer coefficient $\|d_\rho^{\pi^*}/d_\mu^\pi\|_\infty$ is finite although μ puts no weight on Middle, and give a policy for which it is infinite. (c) In Lab 14.1, the stalled policy of the chain has a stationarity measure of about 3×10^{-5} and a mismatch coefficient of about 4×10^5 after 20,000 iterations. What does Lemma 14.1 say about its gap, and why does a tiny gradient not certify near-optimality here?

Exercise 14.5 ●● Softmax policy gradient on two arms

A two-armed bandit has rewards $r = (1, 0.5)$. Run softmax policy gradient with $\eta = 1/3$. (a) From $\theta_0 = (0, 0)$, compute the gradient, θ_1 and π_{θ_1}, and the gap after one step. (b) The gap after 10, 100 and 1,000 iterations is 0.155, 0.0178 and 0.00155. Compare with the bound of Proposition 14.4 and estimate the constant C in a rate C/t. (c) Show that for two arms the best arm's probability never decreases, from any initialization. (d) From $\theta_0 = (-10, 0)$, the best arm needs 66,144 iterations to become the more likely one. Explain this number approximately by studying the logit difference $z = \theta_1 - \theta_2$ while $\pi(a^*)$ is small.

Exercise 14.6 ●● The natural gradient of a softmax policy

For a single state with softmax policy $\pi = \text{softmax}(\theta)$ over K actions and payoffs $q \in \mathbb{R}^K$, let $f(\theta) = \pi^\top q$. (a) Show that the Fisher matrix is $F = \text{diag}(\pi) - \pi\pi^\top$ and that $\nabla f = Fq$. (b) Show that the null space of F is spanned by the all-ones vector when $\pi > 0$, and compute $F^\dagger \nabla f$. (c) Conclude that the natural gradient step with step size η gives $\pi_{\text{new}} \propto \pi e^{\eta q}$, and compare with the ordinary gradient step on θ.

Exercise 14.7 ●● Programming: checking the NPG bound

Using `rlcourse.pg_theory`, run NPG on random MDPs with rewards in $[0, 1]$ (`random_mdp01`) for $\eta \in \{0.01, 0.1, 1, 10\}$ and $T \le 1000$. (a) Check that the gap never exceeds the bound of Theorem 14.6 and that the values never decrease. (b) How loose is the bound? (c) Repeat with step sizes growing as $\eta_t = 0.1/\gamma^t$ and estimate the convergence rate. (d) Which assumption of the theorem would you violate to break monotone improvement, and what happens?

Exercise 14.8 ●● How large is the bias of regularization?

(a) Prove the two-sided inequality $V^\pi(\rho) \le V_\tau^\pi(\rho) \le V^\pi(\rho) + \tau \log|\mathcal{A}|/(1-\gamma)$ used in Proposition 14.8. (b) On the soft Trail of Section 13.5, the regularized optimal policy loses 0.080 of return at $\tau = 0.5$ and 0.407 at $\tau = 1$. Compare with the bound. (c) For a single state with K actions in a continuing problem, one of which has reward 1 and the others $1 - \Delta$, find the bias as a function of Δ, K, τ and γ, and discuss how the bound's dependence on τ, $\log K$ and $1/(1-\gamma)$ compares with the worst case over Δ.

Exercise 14.9 ●● The largest step is soft policy iteration

(a) Show that (14.4) with $\eta = (1-\gamma)/\tau$ is the update $\pi_{t+1} \propto \exp(Q_\tau^{\pi_t}/\tau)$. (b) Prove that this soft policy iteration satisfies $\|Q_\tau^* - Q_\tau^{\pi_t}\|_\infty \le \gamma^t \|Q_\tau^* - Q_\tau^{\pi_0}\|_\infty$. (Use the soft Bellman optimality operator, its γ-contraction, and the fact that each new policy's soft values dominate one application of that operator to the old ones.) (c) Check that this is the case $\eta = (1-\gamma)/\tau$ of Theorem 14.7.

Exercise 14.10 ●● Why the chain stalls

For the chain of Figure 14.2 with $n = 10$, $\gamma = 0.9$ and quit reward 0.1, consider the uniform initial policy and μ concentrated on the first state. (a) Show that $d_\mu^\pi(i) = 0.1 \times 0.45^i$ for the chain's states $i = 0, \dots, 9$. (b) The advantage of right is negative in states 0 to 4 and positive in states 5 to 9 (it is -0.0087 in state 0 and 0.45 in state 9). Compute the softmax gradient of right in states 0 and 9 and explain the first phase of the stall. (c) Explain, using (14.2), why the stall reinforces itself, and why a uniform μ or natural policy gradient avoids it.

Exercise 14.11 ●●● Policy gradient for linear-quadratic control

Consider the scalar system $x_{t+1} = ax_t + bu_t$ with cost $\sum_t (qx_t^2 + ru_t^2)$, no discounting, and policy $u_t = -kx_t$, with $\mathbb{E}[x_0^2] = 1$. (a) Show that $C(k) = (q + rk^2)/(1 - (a - bk)^2)$ for $|a - bk| < 1$, and compute $C'(k)$. (b) For $a = b = q = r = 1$, find the optimal gain from the Riccati equation and check the formula $\nabla C(k) = 2\big((r + b^2 P_k)k - bP_k a\big)\Sigma_k$ at $k = 0.3$, with $P_k = C(k)$ and $\Sigma_k = 1/(1 - (a - bk)^2)$. (c) Estimate numerically the smallest κ with $C(k) - C(k^*) \le \kappa\, C'(k)^2$ on the stabilizing interval. (d) In two dimensions with $A = B = I$, show that $K_1 = \begin{pmatrix} 1 & 0 \\ -10 & 1 \end{pmatrix}$ and $K_2 = \begin{pmatrix} 1 & -10 \\ 0 & 1 \end{pmatrix}$ are stabilizing but their average is not, so the set of stabilizing gains is not convex. Why does gradient descent still converge?

Exercise 14.12 ●●● Two geometries of policy mirror descent

(a) Show that policy mirror descent with the squared Euclidean distance is the projected Q-ascent $\pi_{t+1}(\cdot \mid s) = \mathrm{Proj}_\Delta\big(\pi_t(\cdot \mid s) + \eta Q^{\pi_t}(s, \cdot)\big)$, and with the KL divergence it is NPG. (b) From the uniform Trail policy, take one step of each with $\eta = 0.1$ (for NPG, the step of (14.3) with $\eta = 0.1$). Compare the new policies and their values. (c) Show that both updates become policy iteration as $\eta \to \infty$. (d) Give two reasons to prefer the KL geometry for softmax policies.

Exercise 14.13 ●●● Transfer error

Natural policy gradient with function approximation fits the advantages by regression under the current policy's state distribution, but the analysis needs the fit to be good under the optimal policy's distribution. (a) Construct a small MDP and a feature map for which the fit has zero error on the current policy's distribution and a large error on the optimal policy's distribution. (b) Explain how the relative condition number of A. Agarwal et al. (2021) quantifies this, and

how an exploratory start distribution helps. (c) Relate the example to the role of exploration in the methods of Cai et al. (2020) and Agarwal, Henaff, et al. (2020).

Exercise 14.14 ●●● Research: theory and the practice of PPO

Make a list of the differences between the exact methods analyzed in this chapter (exact NPG, policy mirror descent, regularized NPG) and PPO as it is used in practice (Algorithm 12.2 and its implementation details). For each difference, say which step of the proofs it breaks, what is known about it, and propose a small experiment, on the Trail or on a random MDP, that would measure its effect.

Part V

Models, Exploration, and Data

Three questions decide whether reinforcement learning works in practice. Can the agent learn a model of its world and plan with it (Chapter 15)? How should it gather information it does not yet have (Chapter 16)? And can it learn from data that someone else collected, from logged interactions (Chapter 17) or from demonstrations of good behavior (Chapter 18)?

CHAPTER 15

Model-Based Reinforcement Learning

WHAT YOU WILL LEARN

- Explain what a learned model buys an agent (sample efficiency, lookahead, reuse for new goals) and what it costs (model bias, compounding errors, computation), and place the main families of methods in one taxonomy.
- Estimate a tabular model from data, plan in it by certainty equivalence, and recognize why untried actions need optimism.
- Train an ensemble of neural dynamics models, separate aleatoric from epistemic uncertainty, and evaluate a model by its multi-step error.
- Implement model-predictive control with random shooting, the cross-entropy method and MPPI, and an MBPO-style learner with short branched rollouts.
- State and prove the simulation lemma, and use it to bound the loss of planning in a wrong model (●●).
- Describe latent world models (World Models, PlaNet, Dreamer, TD-MPC) and planning with learned models in games (AlphaZero, MuZero), including value equivalence (●●).
- Summarize what is known about the sample complexity of model-based RL and the open questions about when learned models beat model-free methods (●●●).

A model-free agent learns what to do by doing it. Every improvement in its policy is paid for with real interaction: a robot falls thousands of times before it walks, and a game-playing program plays millions of games. When interaction is cheap, as in a fast simulator, this is fine. When it is expensive, slow or dangerous, as with a real robot, a chemical plant or a patient, it is often impossible.

A *model* changes the bargain. If the agent can predict what the environment will do, it can try actions in its head before trying them for real: it can plan a swing-up of a pendulum before moving a motor, or look many moves ahead in a game before playing one. Chapter 8 introduced this idea in the tabular setting, with Dyna, prioritized sweeping and Monte Carlo tree search, and Chapter 4 showed how to plan exactly when the model is known. This chapter is about what happens when the model must be *learned*, usually by a neural network, from limited data, and is therefore wrong in ways the agent does not know. The central difficulty is easy to state. A planner searches for the actions that the model says are best, and that search is drawn to exactly the places where the model is too optimistic.

We start with what a model buys and costs, and a taxonomy of the ways a model can enter an agent (Section 15.1), illustrated by the simplest model-based method of all: count, estimate, and plan as if the estimate were true. Section 15.2 covers learning models: deterministic and

probabilistic networks, ensembles and the two kinds of uncertainty, and why a model must be judged by its multi-step predictions. Section 15.3 (●●) proves the simulation lemma, which turns model error into value error. Then come the three main ways to use a learned model in deep RL: planning at every step with model-predictive control (Section 15.4); training a model-free learner on short model rollouts, as in MBPO (Section 15.5); and learning a compact latent world model and acting in imagination, as in Dreamer (Section 15.6). Section 15.7 turns to games, where AlphaZero and MuZero combine learned models with tree search. Section 15.8 (●●●) asks how much data a model-based learner needs. On a first reading, the ● sections (Section 15.1, 15.2, 15.4, 15.5, 15.6 and 15.7, without their ●● subsections) form a complete introduction.

15.1 What a model buys and what it costs

A *model* of an environment is anything that predicts how the environment responds to actions. In a finite Markov decision process (MDP, Definition 3.1) the full model is the transition probability $P(s' \mid s, a)$ and the expected reward $r(s, a)$; a learned model is an estimate $\hat{P}$, $\hat{r}$. A *distribution model* gives the probabilities themselves; a *sample model* only produces samples $s' \sim \hat{P}(\cdot \mid s, a)$, which is all most planners need (Chapter 8). In continuous or high-dimensional problems a model is usually a neural network that maps a state and an action to a prediction of the next state and the reward, or to a distribution over them.

What a model buys. The first benefit is *sample efficiency*. Every real transition can be used many times: to fit the model, and then, through the model, to update a value function or a policy at many states and actions that were never visited with exactly that action. The second is *lookahead*. An agent with a model can evaluate the consequences of an action before taking it, which is how chess programs and many robot controllers work. The third is *reuse*. A model of the dynamics does not depend on the reward, so the same model can be used to plan for a new goal without new interaction. A model can also be checked: a controller can be tested in the model before it is trusted on a real system.

What a model costs. Models are wrong, and the errors do not stay small. A model that predicts one step ahead with 1% error may be useless twenty steps ahead, because each prediction is fed back as the input of the next one and the errors compound (Section 15.2). Worse, a planner actively looks for actions that the model rates highly, and some of those ratings are high only because the model is wrong; the planner *exploits* the model. Finally, planning costs computation at every step, and learning a model that is accurate where it matters is itself hard: a model trained to predict every pixel of a video game spends most of its capacity on details that do not affect the score. This last point, that the training objective of a model need not match its use, is called *objective mismatch* (Lambert et al. 2020).

> **KEY IDEA**
>
> A model is only as useful as its accuracy *where the planner looks*. Model-based methods differ mostly in how they keep the planner from exploiting the places where the model is wrong: short rollouts, replanning after every step, uncertainty penalties, or models that predict only what decisions need.

A taxonomy. Figure 15.1 shows the four main ways a learned model can enter an agent. (A) It can *generate experience*: simulated transitions train an ordinary model-free learner, as in Dyna (Algorithm 8.2) and its deep successor MBPO (Section 15.5). (B) It can be used to *plan at decision time*: before every action the agent searches over action sequences in the model, as in model-predictive control (Section 15.4) and in Monte Carlo tree search (Section 8.5.3; Section 15.7). (C) It can *improve learning targets*: a few model steps inside a temporal-difference target trade some bias of the value function for some bias of the model (Section 15.5.1). (D) It can be *differentiated*: if the model and the reward are differentiable, the gradient of an imagined return with respect to the policy parameters can be computed by backpropagation through the model, as in PILCO (Deisenroth and Rasmussen 2011), stochastic value gradients (Heess et al. 2015) and Dreamer (Section 15.6).

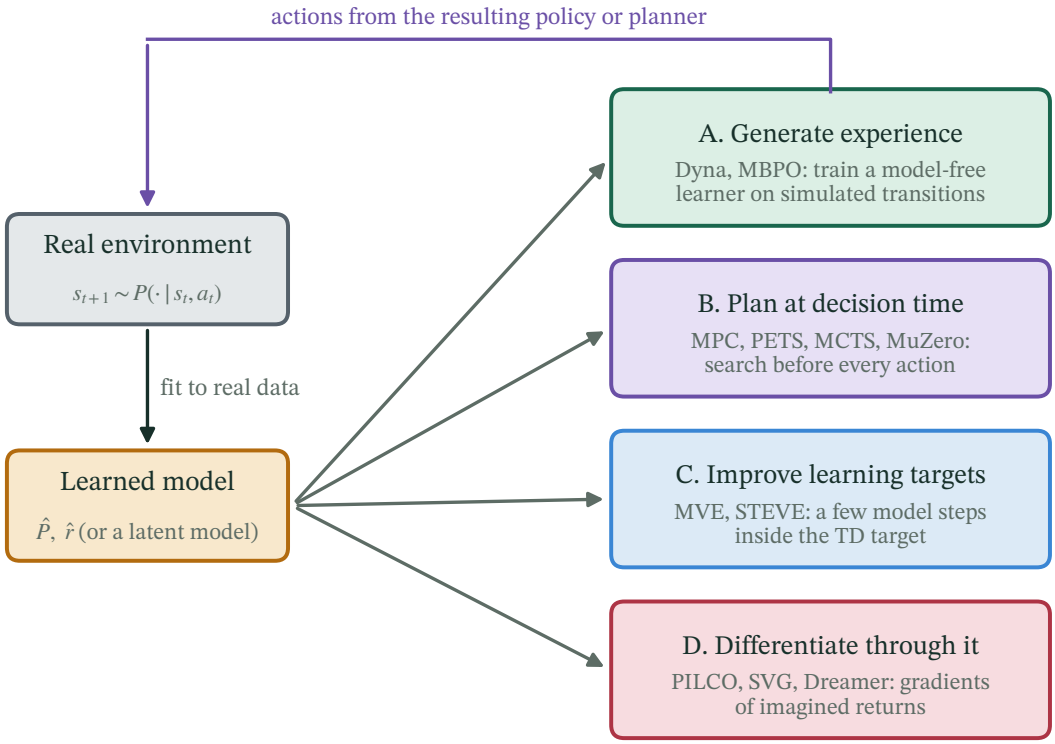


Figure 15.1. Where a learned model can enter an agent. The model is fitted to real transitions and then used in one of four ways: (A) to generate training data for a model-free learner, (B) to search before each action, (C) inside learning targets, or (D) as a differentiable simulator. Many systems combine several; MuZero, for example, plans with a model that was trained only to support planning (B).

Two further distinctions cut across the four uses. The first is *what the model predicts*. Some models predict the next observation itself (a vector of joint angles, or an image); *latent world models* first compress observations into a learned state and predict in that space (Section 15.6); and *value-equivalent* models predict only the quantities that planning needs, such as rewards, values and policies, and may have no recognizable relation to the observations at all (Section 15.7.1). The second is *where the model comes from*. In board games and in many engineering problems the model is known or simulated exactly; elsewhere it must be learned from the agent's own data.

This chapter is mostly about the second case.

15.1.1 The simplest model-based method: certainty equivalence

In a finite MDP, the most natural way to learn a model is to count. Let $N(s,a)$ be the number of times the agent has taken action a in state s, and $N(s,a,s')$ the number of times this led to s'. The *maximum-likelihood model* is

$$\hat{P}(s' \mid s,a) = \frac{N(s,a,s')}{N(s,a)}, \qquad \hat{r}(s,a) = \frac{1}{N(s,a)} \sum_{i\colon (s_i,a_i)=(s,a)} r_i, \tag{15.1}$$

the observed frequencies of next states and the average observed reward. *Certainty equivalence* means planning in the estimated MDP as if it were the true one: solve it by value iteration (Algorithm 4.3) and act greedily with respect to its values. In words: estimate the world, then trust the estimate.

Example 15.1 Certainty-equivalence planning on the slippery Trail

Take the slippery Trail with $\gamma = 0.9$: each go fails with probability 0.2 and leaves the hiker where she is. Its optimal policy goes all the way, with $V^*(\text{Start}) = 3.0839$, $V^*(\text{Middle}) = 3.5122$ and $V^*(\text{Goal}) = 4$ (Chapter 3 shows how to compute these; for example $V^*(\text{Middle}) = 0.9\,(0.8 \cdot 4 + 0.2\,V^*(\text{Middle}))$ gives $V^*(\text{Middle}) = 2.88/0.82$). Suppose an agent that does not know the slip probability has collected 26 transitions:

Table 15.1. A small data set from the slippery Trail and the maximum-likelihood model (15.1) (w.p.: with probability).

State, action	Tried	Outcomes	Model
Start, exit	4	End, reward 1	$\hat{r} = 1$
Start, go	10	7 moved on, 3 stayed	moves on w.p. 0.7
Middle, exit	3	End, reward 2	$\hat{r} = 2$
Middle, go	5	4 moved on, 1 stayed	moves on w.p. 0.8
Goal, finish	4	End, reward 4	$\hat{r} = 4$

Planning in the estimated model is now a small calculation. In Goal, $\hat{V}(\text{Goal}) = 4$. In Middle, going on is worth $\hat{V}(\text{Middle}) = 0.9\,(0.8 \cdot 4 + 0.2\,\hat{V}(\text{Middle}))$, so $\hat{V}(\text{Middle}) = 2.88/0.82 = 3.5122$, which beats exiting (2). In Start, going on is worth $\hat{V}(\text{Start}) = 0.9\,(0.7 \cdot 3.5122 + 0.3\,\hat{V}(\text{Start}))$, so $\hat{V}(\text{Start}) = 2.2127/0.73 = 3.0311$, which beats exiting (1). The plug-in policy goes all the way: it is the optimal policy, although the model is wrong at Start (it thinks go fails 30% of the time instead of 20%). Its estimate of $V^*(\text{Start})$ is 3.0311 instead of 3.0839, an error of 0.0528.

♦

The example shows the typical situation: the model is wrong, the values are slightly wrong, and the policy is still right because the decision was not close. Section 15.3 (●●) makes this precise.

Its main result, the *simulation lemma*, says in words that if every transition distribution of the model is within ϵ_P of the truth and every reward within ϵ_r, then the value of every policy in the model is within $\epsilon_r/(1-\gamma) + \gamma\,\epsilon_P\,r_{\max}/(1-\gamma)^2$ of its true value, where $r_{\max}$ bounds the size of the rewards. For the Trail data the bound is 72, a thousand times the actual error: it is a worst-case guarantee, and the $1/(1-\gamma)^2$ factor is the price of guaranteeing it for every MDP.

PITFALL

An untried action has no estimate, and the default you choose for it decides what the agent learns. Remove the five (Middle, go) transitions from Table 15.1. If the agent models the untried pair pessimistically, as a self-loop with reward 0, then planning gives $\hat{Q}(\text{Middle}, \text{go}) = 1.8 < 2$: it exits at Middle, never tries going on, and settles for a policy worth 1.756 instead of 3.084. If it models the untried pair optimistically, as ending with the largest conceivable return $r_{\max}/(1-\gamma) = 40$, it goes on, observes what happens, and corrects itself. Certainty equivalence without optimism does not explore; optimism in the face of uncertainty, as in R-max (Algorithm 16.1), is the fix (Chapter 16).

CHECKPOINT

1. In Table 15.1, what would $\hat{P}(\text{Middle} \mid \text{Start}, \text{go})$ be if the hiker had stayed 2 times instead of 3 (and moved on 7 times)?
2. Why is the plug-in estimate $\hat{V}(\text{Middle})$ exactly equal to $V^*(\text{Middle})$ in Example 15.1?
3. Does the error in $\hat{V}(\text{Start})$ come from the model at Start, at Middle, or both?

Answers.

1. $7/9 \approx 0.778$.
2. Everything that matters from Middle on, the success probability 0.8 and the rewards, is estimated exactly by the data.
3. Only from Start: the model is exact from Middle on, so the error is the effect of the wrong slip probability at Start.

15.2 Learning models

Counting works for small tables. For a robot arm, a pendulum or a video game, the agent must *generalize*: predict the outcome of actions in states it has never seen exactly, which means fitting a function. This section covers the choices that matter in practice. We use the Pendulum of `rlcourse.envs` as the running example: the observation is $o = (\cos\theta, \sin\theta, \dot{\theta})$, where θ is the angle from upright and $\dot{\theta}$ the angular velocity; the action is a torque $u \in [-2, 2]$; and the reward is $-(\theta^2 + 0.1\,\dot{\theta}^2 + 0.001\,u^2)$, so the best place to be is upright and still.

Deterministic and probabilistic models. A *deterministic model* is a function $\hat{f}_\psi(s, a)$ with parameters ψ, trained by least squares on real transitions (s, a, s') from a data set $\mathcal{D}$:

$$\min_{\psi} \sum_{(s,a,s') \in \mathcal{D}} \left\| \hat{f}_\psi(s, a) - s' \right\|^2 .$$

A *probabilistic model* predicts a distribution, typically a Gaussian with mean $\mu_\psi(s, a)$ and a diagonal covariance with variances $\sigma^2_{\psi,j}(s, a)$, and is trained by maximizing the likelihood of the

observed next states, that is, by minimizing the negative log-likelihood

$$\sum_{(s,a,s')\in\mathcal{D}} \sum_{j} \Big[\frac{(s'_j - \mu_{\psi,j}(s,a))^2}{2\sigma^2_{\psi,j}(s,a)} + \tfrac{1}{2} \log \sigma^2_{\psi,j}(s,a) \Big].$$

The first term is a squared error weighted by the predicted precision; the second stops the model from claiming infinite variance. A probabilistic model can represent genuine randomness in the environment, which a deterministic one averages away. The reward is learned the same way, as one more output.

Predict small, smooth things. Three practical choices make models much easier to learn. Predict the *change* $s' - s$ rather than s' itself, since the change is small and the identity part is free. *Standardize* inputs and outputs to zero mean and unit variance. And choose a representation that respects the geometry of the problem: for the Pendulum, our model predicts the change of the *angle* (recovered from $\cos\theta$ and $\sin\theta$) and of $\dot\theta$, not the change of the cosine and sine, which are tied together and wrap around. With these choices a network with two hidden layers of 32 units, trained on 400 transitions of random torques, predicts the next angle with a median error of about 0.002 radians.

Two kinds of uncertainty. A model can be uncertain for two different reasons, and it matters which. *Aleatoric uncertainty* is randomness in the environment itself: a die roll, sensor noise, a slippery floor. More data does not remove it; the model should represent it, and a planner should account for it. *Epistemic uncertainty* is the model's ignorance: in regions with little or no data, many models fit the data equally well and disagree about the rest. More data removes it, and a planner should not trust predictions there.

A practical way to estimate epistemic uncertainty is an *ensemble*: train K models (typically $K = 5$) from different random initializations (Lakshminarayanan et al. 2017), often each on its own bootstrap resample of the data (a sample of the same size drawn with replacement), as in PETS (Chua et al. 2018). Where there is plenty of data the members agree; away from the data they extrapolate differently, and their disagreement signals ignorance. With probabilistic members, the two uncertainties can be separated: for a prediction at input x, with member means $\mu_k(x)$ and variances $\sigma^2_k(x)$, the law of total variance gives

$$\underbrace{\operatorname{Var}[y \mid x]}_{\text{total}} = \underbrace{\frac{1}{K}\sum_k \sigma^2_k(x)}_{\text{aleatoric}} + \underbrace{\frac{1}{K}\sum_k \big(\mu_k(x) - \bar\mu(x)\big)^2}_{\text{epistemic}}, \qquad \bar\mu(x) = \frac{1}{K}\sum_k \mu_k(x),$$

for the mixture that picks a member at random. Figure 15.2 shows both on a one-dimensional regression problem.

Ensembles are not the only option. Gaussian processes give principled uncertainty for small data sets and were the model of PILCO, which learned to swing up a real cart-pole from a few trials (Deisenroth and Rasmussen 2011); Bayesian neural networks and dropout are alternatives. Ensembles are popular in deep model-based RL because they are simple, scale well and work reasonably in practice, as in ME-TRPO (Kurutach et al. 2018), PETS (Chua et al. 2018) and MBPO (Janner et al. 2019).

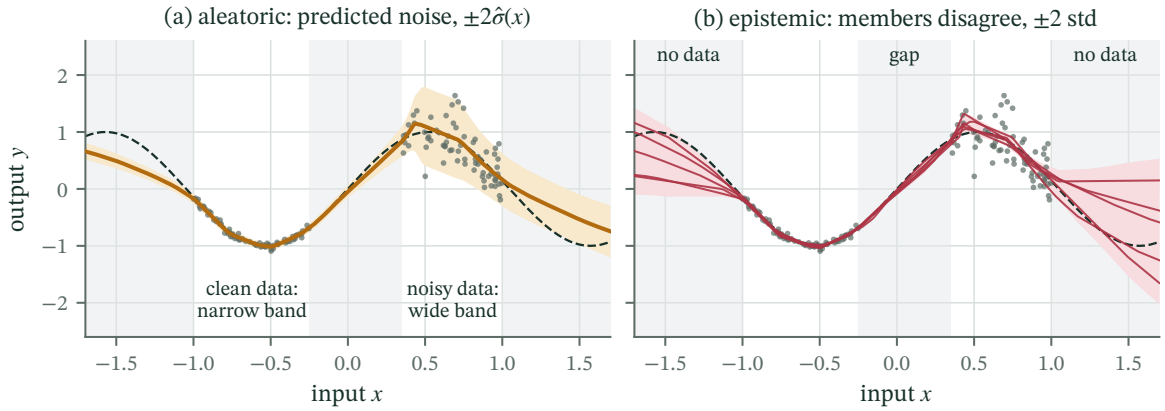


Figure 15.2. Aleatoric and epistemic uncertainty in an ensemble of five probabilistic networks fitted to the dots (true function dashed; shaded columns have no data). (a) The predicted noise band is narrow where the data are clean and wide where they are noisy, and it stays wide however much noisy data we add. (b) The members (thin lines) agree near the data and fan out in the gap and beyond the data; this disagreement shrinks with more data. Note that in the gap the members happen to agree on a wrong curve: disagreement is a useful signal, not a guarantee.

Judge a model by its multi-step predictions. A planner uses a model by chaining it: predict $\hat{s}_1$ from s_0, then $\hat{s}_2$ from $\hat{s}_1$, and so on. Small one-step errors then compound, for two reasons. The errors accumulate, and, more insidiously, the model is evaluated on its own predictions, which drift into states unlike those it was trained on. Figure 15.3 measures this on the Pendulum. A model trained on 200 random steps has a median one-step angle error of 0.006 radians, but after 50 steps of open-loop prediction (the model feeding on its own outputs, with the true action sequence) the error is 0.23 radians. Eight times more data cuts both errors roughly fivefold on test data from the same random policy. But on test data from a swing-up controller, which visits upright and fast states that random torques rarely reach, even the better model's predictions are off by a radian or more within 20 to 30 steps: near the upright position the pendulum is unstable, and tiny errors in the predicted state are amplified as in the real system.

PITFALL

Low one-step error does not make a model good for planning. The one-step error is measured on states from the data; planning queries the model on its own predictions and on the states the planner prefers. Evaluate a model the way it will be used: by open-loop error over the planning horizon, on data from the policy that will use it, and by the gap between the returns it predicts and the returns actually obtained.

Latent models. When observations are images, predicting the next image pixel by pixel is expensive and mostly beside the point. *Latent dynamics models* learn an encoder that maps observations to a compact state, and predict in that state. What the latent state must preserve is a central design question: enough to reconstruct the observation (World Models, PlaNet, Dreamer), or only enough to predict rewards and values (MuZero, TD-MPC). Section 15.6 and Section 15.7 return to this.

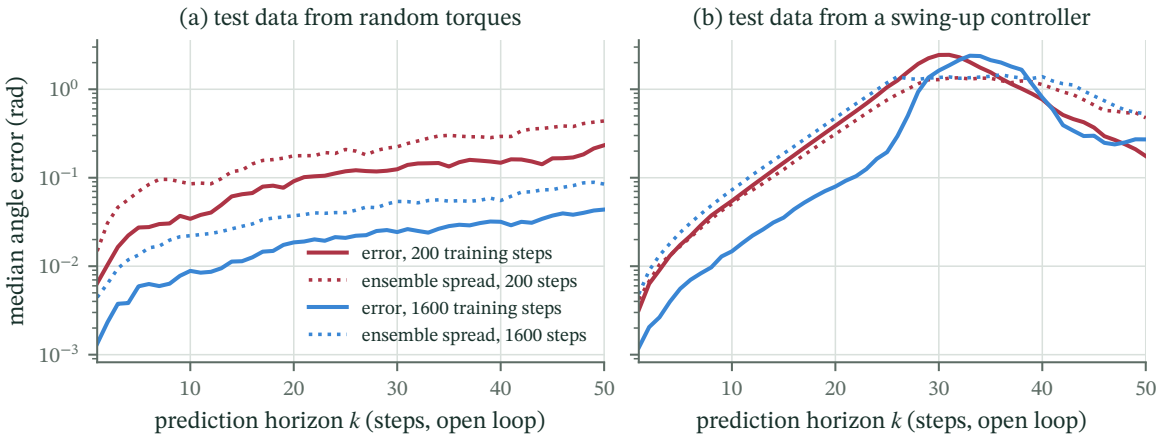


Figure 15.3. Open-loop prediction error of learned Pendulum models (median over 60 test windows, log scale) as a function of how many steps the model predicts ahead, for ensembles trained on 200 and 1600 steps of random torques. Solid lines: error of the ensemble's mean prediction. Dotted lines: the spread of the five members. (a) On test data from the same random policy, errors grow steadily with the horizon and more data helps. (b) On data from a swing-up controller, which visits unfamiliar unstable states, errors grow much faster and reach a radian within 20–30 steps, even for the better model. In both cases the spread rises with the error, so it can warn a planner when to stop trusting a rollout.

LAB

Lab 15.1: Learning models and planning with them (`labs/ch15_model_based.py`, about 60 minutes; the script runs in about 20 seconds). Run `python3 labs/ch15_model_based.py`. Task 1 reproduces Example 15.1 and the pitfall about untried actions; Task 2 checks the simulation lemma on random MDPs; Task 3 trains a Pendulum ensemble and measures one-step versus open-loop error; Tasks 4 and 5 control the Pendulum by MPC and compare model-free, MBPO-style and PETS-style learning.

15.3 The simulation lemma ●●

How wrong can the values computed in a model be? The answer, the simulation lemma, is the basic tool of the theory of model-based RL. Its proof is short, and its form explains much of the practice: value errors scale with the model error times the *effective horizon* $1/(1-\gamma)$, once for the reward and twice for the transitions, because a transition error at one step affects all the rewards after it.

Setting. Let $M = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$ be a finite MDP and $\hat{M} = (\mathcal{S}, \mathcal{A}, \hat{P}, \hat{r}, \gamma)$ a model with the same states, actions and discount, $\gamma < 1$. For a policy π, write $P^{\pi}(s, s') = \sum_a \pi(a \mid s) P(s' \mid s, a)$ and $r^{\pi}(s) = \sum_a \pi(a \mid s) r(s, a)$, and similarly $\hat{P}^{\pi}$, $\hat{r}^{\pi}$. The values V_M^{π} and $V_{\hat{M}}^{\pi}$ are the unique solutions of the Bellman equations $V = r^{\pi} + \gamma P^{\pi} V$ in M and in $\hat{M}$ ((3.9)). Assume

$$|\hat{r}(s,a) - r(s,a)| \le \epsilon_r, \qquad \sum_{s'} |\hat{P}(s' \mid s,a) - P(s' \mid s,a)| \le \epsilon_P, \qquad |r(s,a)| \le r_{\max}$$

for all s and a. The second quantity is the L^1 distance between the two next-state distributions,

twice their total-variation distance.

Lemma 15.1 Simulation lemma

Under these assumptions, for every policy π (stationary and possibly stochastic),

$$\left\| V^{\pi}_{\hat{M}} - V^{\pi}_{M} \right\|_{\infty} \le \frac{\epsilon_r}{1-\gamma} + \frac{\gamma\, \epsilon_P\, r_{\max}}{(1-\gamma)^2}.$$

More precisely, the difference satisfies the identity

$$V^{\pi}_{\hat{M}} - V^{\pi}_{M} = (I - \gamma \hat{P}^{\pi})^{-1}\big[(\hat{r}^{\pi} - r^{\pi}) + \gamma\,(\hat{P}^{\pi} - P^{\pi})\, V^{\pi}_{M}\big]. \tag{15.2}$$

Proof. Write $V = V^{\pi}_{M}$ and $\hat{V} = V^{\pi}_{\hat{M}}$. Subtract the two Bellman equations and add and subtract $\gamma \hat{P}^{\pi} V$:

$$\hat{V} - V = (\hat{r}^{\pi} - r^{\pi}) + \gamma \hat{P}^{\pi}\hat{V} - \gamma P^{\pi} V = (\hat{r}^{\pi} - r^{\pi}) + \gamma \hat{P}^{\pi}(\hat{V} - V) + \gamma(\hat{P}^{\pi} - P^{\pi})V.$$

Collect the terms in $\hat{V} - V$ on the left: $(I - \gamma\hat{P}^{\pi})(\hat{V} - V) = (\hat{r}^{\pi} - r^{\pi}) + \gamma(\hat{P}^{\pi} - P^{\pi})V$. The matrix $I - \gamma\hat{P}^{\pi}$ is invertible with $(I - \gamma\hat{P}^{\pi})^{-1} = \sum_{k \ge 0} \gamma^k (\hat{P}^{\pi})^k$, because $\hat{P}^{\pi}$ is a stochastic matrix and $\gamma < 1$; this gives (15.2).

For the bound, note three facts. First, every row of $(\hat{P}^{\pi})^k$ is a probability distribution, so $\|(\hat{P}^{\pi})^k x\|_{\infty} \le \|x\|_{\infty}$ and therefore $\|(I - \gamma\hat{P}^{\pi})^{-1} x\|_{\infty} \le \sum_k \gamma^k \|x\|_{\infty} = \|x\|_{\infty}/(1-\gamma)$. Second, $|\hat{r}^{\pi}(s) - r^{\pi}(s)| \le \sum_a \pi(a \mid s)|\hat{r}(s,a) - r(s,a)| \le \epsilon_r$. Third, for every state s,

$$\begin{aligned} \big|\big((\hat{P}^{\pi} - P^{\pi})V\big)(s)\big| &= \Big|\sum_a \pi(a \mid s) \sum_{s'} \big(\hat{P}(s' \mid s,a) - P(s' \mid s,a)\big) V(s')\Big| \\ &\le \sum_a \pi(a \mid s)\, \epsilon_P\, \|V\|_{\infty} \le \epsilon_P\, \frac{r_{\max}}{1-\gamma}, \end{aligned}$$

where we used $|\sum_{s'} x(s')V(s')| \le \sum_{s'} |x(s')|\, \|V\|_{\infty}$ and $\|V\|_{\infty} \le r_{\max}/(1-\gamma)$, which holds because every return in M is a discounted sum of rewards bounded by $r_{\max}$. Putting the three facts together,

$$\|\hat{V} - V\|_{\infty} \le \frac{1}{1-\gamma}\Big(\epsilon_r + \gamma\, \epsilon_P \frac{r_{\max}}{1-\gamma}\Big),$$

which is the claim. ∎

The identity (15.2) reads: the error of the model at each state, measured on the true values, is spread out over the future by the model's own dynamics. On the Trail data of Example 15.1, the model is exact except at (Start, go), where the local error is $\gamma\,[(0.7 - 0.8)\, V(\text{Middle}) + (0.3 - 0.2)\, V(\text{Start})] = 0.9 \times (-0.0428) = -0.0385$. The model returns to Start with probability 0.3 per step under the plug-in policy, so this error is collected $1/(1 - 0.27) = 1.37$ times on average: $-0.0385 \times 1.37 = -0.0528$, exactly the observed error. The bound of the lemma, 72, assumes instead that every state has the worst possible error, all in the same direction, for $1/(1-\gamma)^2 = 100$ effective steps.

Two refinements are worth knowing. Because $\sum_{s'} (\hat{P} - P)(s' \mid s,a) = 0$, any constant can be subtracted from V in the third step; so $\|V\|_{\infty}$ can be replaced by half the range $\frac{1}{2}(\max V - \min V)$,

which for rewards in $[0, r_{\max}]$ halves the transition term. And the $1/(1-\gamma)^2$ dependence cannot be improved in general (Exercise 15.2).

Corollary 15.2 The cost of planning in a wrong model

Let $\hat{\pi}$ be an optimal policy of $\hat{M}$ and π^* an optimal policy of M, and let $\Delta = \epsilon_r/(1-\gamma) + \gamma\,\epsilon_P\, r_{\max}/(1-\gamma)^2$. Then for every state s,

$$V_M^{\pi^*}(s) - V_M^{\hat{\pi}}(s) \le 2\Delta.$$

Proof. Decompose

$$V_M^{\pi^*} - V_M^{\hat{\pi}} = \underbrace{\left(V_M^{\pi^*} - V_{\hat{M}}^{\pi^*}\right)}_{\le \Delta} + \underbrace{\left(V_{\hat{M}}^{\pi^*} - V_{\hat{M}}^{\hat{\pi}}\right)}_{\le 0} + \underbrace{\left(V_{\hat{M}}^{\hat{\pi}} - V_M^{\hat{\pi}}\right)}_{\le \Delta}.$$

The first and third terms are bounded by the simulation lemma applied to π^* and to $\hat{\pi}$; the middle term is non-positive at every state because $\hat{\pi}$ is optimal in $\hat{M}$. ■

The corollary is the theoretical justification of certainty equivalence, and also of its limits. It needs the model to be accurate *for every policy*, including the plug-in policy, whose state distribution the model has not seen; the uniform assumption over all (s, a) hides this. In deep RL the model is accurate only near the data, and the planner's policy leaves the data. This is the formal version of model exploitation.

Compounding over a horizon. A complementary view follows the state distribution forward. Let μ_k and $\hat{\mu}_k$ be the distributions of the state after k steps when a fixed policy is run from the same start in M and in $\hat{M}$, and let $\epsilon_{D_{\mathrm{TV}}} = \epsilon_P/2$ bound the total-variation distance of the one-step distributions. Then

$$D_{\mathrm{TV}}(\hat{\mu}_k, \mu_k) \le k\,\epsilon_{D_{\mathrm{TV}}}. \tag{15.3}$$

The proof is a telescoping argument: run the model for the first j steps and the true MDP for the remaining $k-j$; consecutive hybrids differ only in one step, which moves at most $\epsilon_{D_{\mathrm{TV}}}$ of probability, and later steps, being the same Markov kernel applied to both, cannot increase the total-variation distance. So distributional errors grow at most linearly in the horizon. A deterministic model of a deterministic system is different: if the model has one-step error at most ϵ and the true dynamics are L-Lipschitz, the prediction error after k steps can be as large as $\epsilon\,(L^k - 1)/(L-1)$, which grows *exponentially* when $L > 1$, as near the upright Pendulum (Exercise 15.5). Both views say the same thing at different strengths: long model rollouts are dangerous.

Objective mismatch. The simulation lemma also explains why the usual training loss of a model, the one-step likelihood on the data distribution, is not the right objective. What matters for control is the error $(\hat{P} - P)V$ weighted by the states the planner visits, and in directions that change values. Lambert et al. (2020) found that the likelihood of learned dynamics models correlates only weakly with the performance of the controllers built on them. *Value-aware* model learning trains the model to make $(\hat{P} - P)V$ small for the relevant value functions instead (Farahmand et al. 2017), an idea taken to its conclusion by value-equivalent models (Section 15.7.1).

15.4 Planning at every step: model-predictive control

The most direct way to use a model is to plan with it before every action. *Model-predictive control* (MPC), also called *receding-horizon control*, has been used in industrial process control for decades. At every time step it optimizes a sequence of H actions in the model, executes only the first one, observes the real next state, and optimizes again. Because the plan is recomputed from the real state at every step, model errors cannot accumulate over more than one step of real execution; the controller corrects itself continually. And because only a short horizon is simulated, the long-horizon compounding of Section 15.2 is avoided.

Formally, at state s_t MPC (approximately) solves

$$\max_{a_t,\dots,a_{t+H-1}} \mathbb{E}_{\hat{M}}\Big[\sum_{h=0}^{H-1} \gamma^h \hat{r}(\hat{s}_{t+h}, a_{t+h}) + \gamma^H \hat{V}(\hat{s}_{t+H})\Big], \qquad \hat{s}_t = s_t, \quad \hat{s}_{t+h+1} \sim \hat{P}(\cdot \mid \hat{s}_{t+h}, a_{t+h}),$$

where $\hat{V}$ is an optional *terminal value* that estimates what happens after the horizon (zero if none is available). The optimization is over action sequences, not policies, so it needs no value function and no gradient of the model: any black-box optimizer will do.

Algorithm 15.1 Model-predictive control with the cross-entropy method

---

Input: a model ($\hat{P}$, $\hat{r}$, possibly an ensemble), horizon H, population size N, number of elites m, iterations J, initial standard deviation σ_0; optionally a terminal value $\hat{V}$.

1. Initialize the plan $\mu = (\mu_0, \dots, \mu_{H-1}) \leftarrow 0$.
2. For each time step t:
   1. Observe s_t; set $\sigma \leftarrow \sigma_0$ (for every component).
   2. Repeat J times:
      1. Sample N action sequences $U^{(i)} \sim \mathcal{N}(\mu, \operatorname{diag}\sigma^2)$, clipped to the action bounds.
      2. Simulate each in the model from s_t and compute its predicted return $\hat{G}(U^{(i)})$ (with an ensemble: average over members).
      3. Let $\mathcal{D}_{\text{elite}}$ be the m sequences with the highest predicted returns; set μ and σ to their mean and standard deviation.
   3. Apply $a_t = \mu_0$ in the real environment; observe r_t and s_{t+1}; store the transition (and refit the model from time to time).
   4. Shift the plan: $\mu \leftarrow (\mu_1, \dots, \mu_{H-1}, 0)$.

Output: the actions $a_0, a_1, \dots$ (MPC is a controller, not a learned policy).

The inner loop is the *cross-entropy method* (CEM) (de Boer et al. 2005): sample candidates from a Gaussian, keep the best, refit the Gaussian to them, repeat. Each iteration moves the sampling distribution toward better sequences and shrinks it; Exercise 15.9 shows that the refit is the maximum-likelihood fit to the elites, which is where the name comes from. Shifting the previous plan by one step (step 2.4) *warm-starts* the next optimization, so that a few iterations per step suffice.

Two simpler and one smoother optimizer are common. *Random shooting* samples NJ sequences uniformly and keeps the best; it is trivially parallel and was used with neural network dynamics by Nagabandi et al. (2018), but it scales badly with the horizon, because a random sequence of 30 actions is almost never a good one. *MPPI* (model-predictive path integral control) (Williams et al. 2017) replaces the hard elite set by soft weights: with costs $C_i = -\hat{G}(U^{(i)})$ and a temperature $\lambda > 0$,

$$\mu \leftarrow \sum_{i=1}^{N} w_i \, U^{(i)}, \qquad w_i = \frac{\exp\bigl(-(C_i - \min_j C_j)/\lambda\bigr)}{\sum_j \exp\bigl(-(C_j - \min_l C_l)/\lambda\bigr)}, \tag{15.4}$$

a weighted average that tilts toward low-cost sequences; small λ approaches "take the best", large λ the plain average. Exercise 15.10 derives these weights from a KL-regularized optimization.

MPC on the Pendulum. Start the pendulum hanging straight down and at rest. The torque limit is too weak to lift it directly, so a good controller must pump: swing back and forth to gain energy, then catch the pendulum at the top. Figure 15.4 shows MPC doing exactly that. Panel (a) shows one optimization: from the hanging state, with horizon $H = 30$ and 100 sampled sequences per iteration, CEM lowers the best predicted cost from 256 after one iteration to 228 after ten, while random shooting with the same number of samples reaches only 254. Panel (b) shows the closed loop with $H = 15$ and three CEM iterations per step: with the exact simulator CEM pumps twice and balances the pendulum within about 4 seconds (return −387); random shooting needs longer (return −496). With an ensemble *learned* from only 400 steps of random torques, CEM does almost as well as with the exact simulator (return −397).

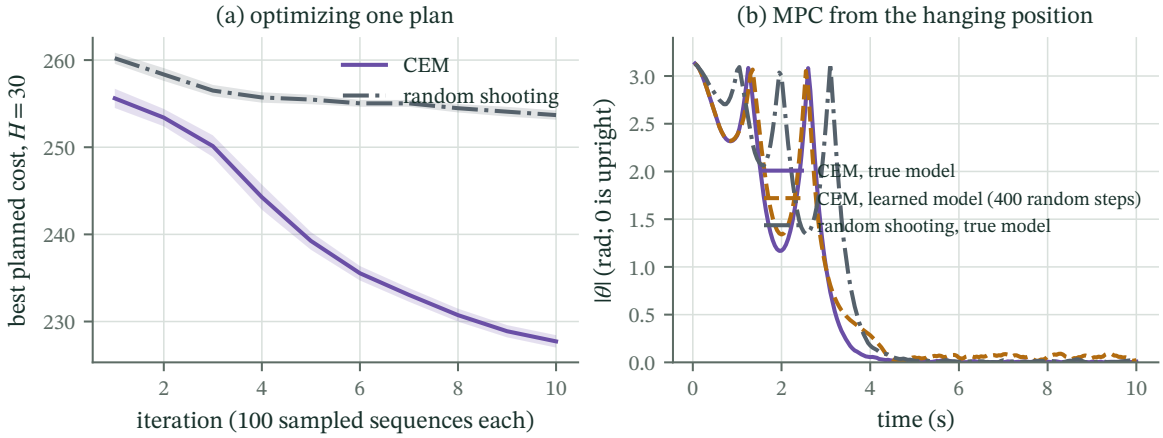


Figure 15.4. Model-predictive control on the Pendulum. (a) Optimizing a single 30-step plan from the hanging position: the best predicted cost after each iteration (mean over 20 runs; the bands of one standard error are thin). CEM keeps improving; random shooting with the same samples stalls. (b) The closed loop from the hanging position, replanning 15 steps ahead at every step: $|\theta|$ falls to zero when the pendulum is upright. With the exact simulator, CEM pumps twice and balances; random shooting takes longer; and a model learned from 400 random steps is nearly as good as the exact simulator.

PETS. Chua et al. (2018) combined MPC with the ensembles of Section 15.2 in an algorithm called PETS (probabilistic ensembles with trajectory sampling). The model is an ensemble of probabilistic networks; each candidate action sequence is evaluated by propagating several *par-*

ticles through the model, each assigned to an ensemble member either for the whole rollout or anew at each step; and CEM optimizes the average predicted return. The agent alternates between acting by MPC and refitting the model on all data so far. PETS was reported to reach the final performance of strong model-free methods on several continuous-control benchmarks with many fewer samples. Our small version, `run_pets` in the lab, swings up the Pendulum in its second episode, after only 200 steps of random data (Figure 15.5).

PITFALL

The optimizer will find the model's mistakes. A powerful optimizer with a learned model is an error-seeking device: if some action sequence is predicted to do much better than it really does, the optimizer will find and prefer it. Symptoms are a large gap between predicted and realized returns, and plans that get worse when the optimizer is given more samples or a longer horizon. Remedies are short horizons, replanning, averaging over an ensemble (which discounts sequences that only some members like), explicit uncertainty penalties, and collecting new data where the planner went.

MPC has two costs. It needs a lot of computation *at decision time*, a full optimization per action, which is a problem for fast control loops unless the model is cheap or the computation is parallel. And its horizon is short, so it is myopic unless a terminal value $\hat{V}$ summarizes the future; learning such a value is the bridge to the methods of the next sections and to TD-MPC (Section 15.6.1).

15.5 Learning a policy from model rollouts: MBPO

The opposite way to use a model is to learn a policy from it ahead of time, so that acting is cheap. Dyna (Algorithm 8.2) does this in tables: after each real step it makes several Q-learning updates on transitions sampled from the learned model. The deep-RL version replaces the table by an ensemble of neural networks and Q-learning by an off-policy deep learner such as SAC (Algorithm 13.3). The question is how to generate model data without feeding the learner the model's mistakes.

One extreme would train the policy entirely inside the model, on long imagined trajectories from the start states. The simulation lemma says why this is risky: the value error grows with the model error times $1/(1-\gamma)^2$, and long rollouts compound. The other extreme, rollouts of length zero, is plain model-free learning from replayed data. *Model-based policy optimization* (MBPO) (Janner et al. 2019) chooses a middle way: start many *short* rollouts, of k steps (often 1 to 5), from states sampled from the *real* replay buffer, following the current policy in the model. The start states have the right distribution because they are real; the short rollouts keep the compounding small; and the policy's own actions at those states, which the real data never contain, are what the model supplies.

Algorithm 15.2 Model-based policy optimization (MBPO)

Input: rollout length k, rollouts per real step M, updates per real step G, fraction β of real data in each batch (often small or zero), ensemble size K.

1. Initialize the policy (and critic) of an off-policy learner, a model ensemble $\{\hat{P}_{\psi_1}, \dots, \hat{P}_{\psi_K}\}$, a real data set $\mathcal{D}_{\text{env}}$ and a model data set $\mathcal{D}_{\text{model}}$.
2. For each epoch (for example, every episode or every 1000 steps):
    1. Train the ensemble on $\mathcal{D}_{\text{env}}$ by maximum likelihood.
    2. For each real environment step:
        1. Act with the current policy; add the transition (s, a, r, s') to $\mathcal{D}_{\text{env}}$.
        2. For M rollouts: sample a state s uniformly from $\mathcal{D}_{\text{env}}$; follow the current policy for k steps in the model, using a randomly chosen ensemble member for each step; add the transitions to $\mathcal{D}_{\text{model}}$.
        3. For G updates: update the learner on a batch drawn from $\mathcal{D}_{\text{model}}$ (and a fraction β from $\mathcal{D}_{\text{env}}$).

Output: the learned policy.

The number of updates per real step G, often called the *update-to-data ratio*, is large in MBPO, often 20 or more with SAC, compared with one in standard SAC. This is where the sample efficiency comes from: each real transition supports many updates, and the model makes those updates useful by supplying fresh, on-policy transitions instead of the same old replayed ones. Janner et al. (2019) also gave a theoretical argument for short rollouts. Their bound on the true return of a policy trained on branched rollouts has two kinds of error terms: terms from the model error that grow with the rollout length k, and terms from the difference between the data-collecting policy and the current one that shrink with k; under an empirical assumption on how quickly model error grows as the policy changes, the bound favors short but nonzero rollouts. They reported that MBPO matched the final performance of SAC on standard continuous-control benchmarks with a fraction of its samples.

What our experiment shows. Figure 15.5 compares four agents on the Pendulum with five discrete torques, with the same exploration schedule and the same real-data budget. A compact Double DQN (Algorithm 10.1) with one update per real step needs about 2,400 real steps to swing up reliably. The same learner with five updates per step on replayed real data, and an MBPO-style agent with five updates per step mostly on 3-step rollouts of a learned ensemble, both first swing up after about 1,200 steps and do so consistently after 1,600 to 2,000. The PETS-style planner, which uses its model at decision time, gets there in 400.

The striking result is the middle one: on this problem, the learned model adds little beyond what replaying real data more often achieves. This is not a quirk. A replay buffer *is* a model, a nonparametric one that reproduces past transitions exactly; Hasselt et al. (2019) argued that a parametric model used only to generate more of the same kind of data must generalize better than replay to help, and showed that a data-efficient version of Rainbow, with more updates per real step, performed as well as or better than the model-based SimPLe agent (Kaiser et al. 2020) on the Atari 100k benchmark. In continuous control, X. Chen et al. (2021) reported that a model-free method designed for high update-to-data ratios (REDQ) matched MBPO's sample efficiency. Models pay off most clearly when they are used for something replay cannot do: planning at decision time, as with MPC, or predicting the consequences of actions and situations that are not in the data.

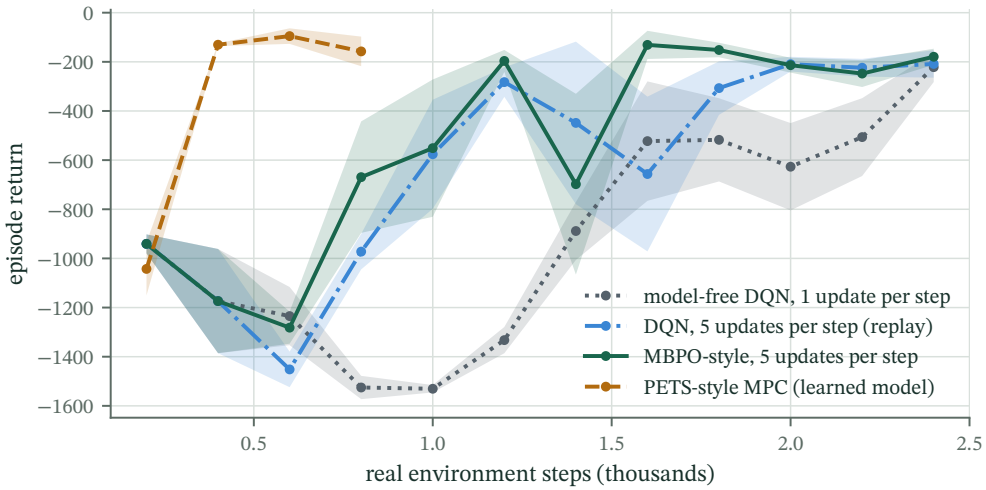


Figure 15.5. Sample efficiency on the Pendulum (mean episode return over 4 seeds; bands are one standard error). Model-free DQN with one update per real step is slowest. Doing five updates per real step, either on replayed real data or mostly on 3-step model rollouts (MBPO-style), learns about twice as fast; on this small problem the two are within noise of each other. Planning at decision time with a learned model (PETS-style MPC) swings up after a single episode of random data.

PITFALL

Model data are not new data. Synthetic transitions contain no information about the environment beyond what is in the real data and in the model's inductive biases. They can spread that information to new states and actions, and they let a learner make more updates per real step, but a model fitted to a narrow data set will happily generate a large, confident and wrong data set. Watch the gap between the returns the model predicts and those the real environment delivers.

15.5.1 Model-based value expansion ●●

A third use of the model sits between the two. Instead of generating whole transitions for the learner, the model is used only inside the *target* of a temporal-difference update. For a real transition (s_t, a_t, r_t, s_{t+1}), *model-based value expansion* (MVE) (Feinberg et al. 2018) rolls the model out for H steps from the real next state, $\hat{s}_{t+1} = s_{t+1}$, with actions $\hat{a}_{t+i}$ from the current policy, and uses the target

$$y_t^{(H)} = r_t + \sum_{i=1}^{H} \gamma^i \, \hat{r}(\hat{s}_{t+i}, \hat{a}_{t+i}) + \gamma^{H+1} \, \bar{Q}(\hat{s}_{t+H+1}, \hat{a}_{t+H+1}), \tag{15.5}$$

where $\bar{Q}$ is the target network of the critic. With $H = 0$ this is the ordinary one-step target; with larger H more of the target comes from the model and less from the critic. If the model were perfect, the critic's error would enter the target multiplied by γ^{H+1} instead of γ, which helps early in training when the critic is poor. If the model is wrong, its errors enter through H imagined rewards. The best H therefore depends on which of the two is currently more accurate, and it changes during training and from state to state.

STEVE (stochastic ensemble value expansion) (Buckman et al. 2018) chooses the horizon automatically. It computes the targets $y^{(0)}, \dots, y^{(H)}$ for every combination of model, reward and critic ensemble members, measures the mean and variance of each $y^{(h)}$ across the ensemble, and combines the means with weights proportional to the inverse variances. Where the model members disagree, the long-horizon targets get little weight and the target falls back to the critic. This is the inverse-variance weighting of independent estimates, applied to the bias-variance trade-off between model and critic, a cousin of the λ-return ((7.9)).

15.6 Latent world models

When observations are images, neither MPC with an image-predicting model nor MBPO is practical. *Latent world models* learn a compact state in which dynamics are easy to predict, and plan or learn behavior there. The line of work from World Models to DreamerV3 is one of the main successes of model-based deep RL.

World Models. Ha and Schmidhuber (2018) trained three components separately on car-racing and first-person-shooter video games: a variational autoencoder that compresses each frame into a small latent vector z_t; a recurrent network with a mixture-density output that predicts the distribution of z_{t+1} from z_t, the action and its own hidden state h_t; and a tiny linear controller that maps (z_t, h_t) to actions, optimized by an evolution strategy. Strikingly, a controller trained entirely inside the model's "dream" could transfer back to the real game, provided the dream was made noisier than the real game (by a temperature parameter) so that the controller could not exploit the model's mistakes.

PlaNet and the recurrent state-space model. Hafner et al. (2019) introduced the *recurrent state-space model* (RSSM), which became the backbone of the Dreamer family. Its state has a deterministic part h_t, computed by a recurrent network, and a stochastic part z_t:

$$h_t = f_\psi(h_{t-1}, z_{t-1}, a_{t-1}), \qquad z_t \sim q_\psi(z_t \mid h_t, o_t)\ \text{(posterior)}, \qquad \hat{z}_t \sim p_\psi(\hat{z}_t \mid h_t)\ \text{(prior)}.$$

The *posterior* uses the current observation and is used when real observations are available; the *prior* predicts without it and is what the model uses to imagine the future. Training maximizes a variational bound: reconstruct the observation and the reward from (h_t, z_t), and keep the prior close to the posterior in KL divergence, so that imagination resembles reality. PlaNet then planned in the latent space with CEM, an MPC agent that never predicts an image while planning, and it was reported to solve continuous-control tasks from pixels with far fewer episodes than model-free agents of the time.

Dreamer: actor-critic in imagination. Hafner et al. (2020) replaced planning by learning: an actor and a critic are trained entirely on trajectories *imagined* by the RSSM (Figure 15.6a). From latent states inferred from real data, the actor picks actions, the prior predicts the next latent states, and a reward head and a continuation head (the probability $\hat{c}$ that the episode goes on) score them. The critic v is trained to predict λ-returns computed along the imagined trajectory,

$$V_t^\lambda = \hat{r}_t + \gamma\, \hat{c}_t \big[(1-\lambda)\, v(\hat{s}_{t+1}) + \lambda\, V_{t+1}^\lambda\big],$$

the recursive form of the λ-return of (7.9) with the predicted continuation in place of the termination flag. The actor maximizes the same λ-returns; in the first Dreamer, by backpropagating

them through the learned dynamics, an instance of use (D) in the taxonomy; later versions also use score-function (REINFORCE) gradients (Algorithm 11.1), which also work for discrete actions. Because imagination is cheap, each training step imagines short trajectories (15 steps in the paper) from every latent state of a batch of real sequences, thousands of them at once. DreamerV2 (Hafner et al. 2021) used discrete (categorical) latent variables and was reported to reach human-level performance on the Atari benchmark by learning behavior inside its world model.

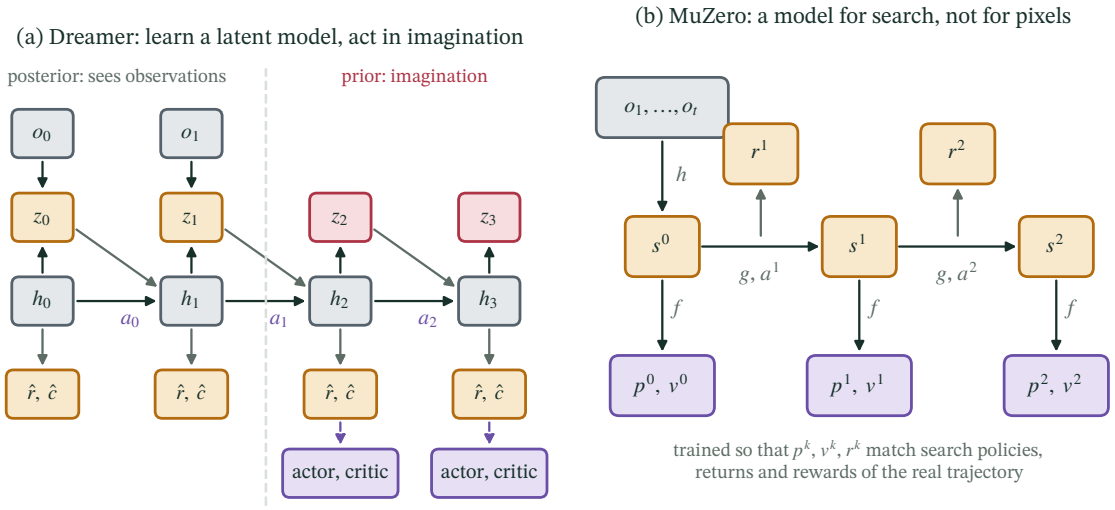


Figure 15.6. Two latent world models. (a) Dreamer: on real data (left of the dashed line) the posterior state z_t is inferred from the deterministic state h_t and the observation o_t; in imagination (right) the prior predicts z_t from h_t alone, the heads predict rewards and continuation, and the actor and critic learn from the imagined states. (b) MuZero: a representation function h encodes the observation history into s^0; a dynamics function g maps a state and an action to a reward and the next state; a prediction function f outputs a policy and a value. Nothing forces s^k to resemble an observation: the model is trained only to make its rewards, values and policies match real ones.

DreamerV3 and robustness. Deep RL algorithms are notoriously sensitive to the scale of rewards and observations, so they are usually re-tuned for each domain. Hafner et al. (2025) designed DreamerV3 to work across domains with one fixed set of hyperparameters. Its main tools are worth knowing because they apply far beyond Dreamer. Predictions of quantities with very different scales (observation vectors, rewards, values) are made in *symlog* space, $\mathrm{symlog}(x) = \mathrm{sign}(x)\ln(1+|x|)$, which compresses large magnitudes while staying close to the identity near zero; rewards and values are predicted as distributions over exponentially spaced bins ("two-hot" targets) rather than by squared error. Returns are *normalized* by dividing advantages by the spread between the 5th and 95th percentiles of recent returns, but only when that spread exceeds one, so that large returns are scaled down while small returns are not amplified into noise. The KL terms of the world model are clipped below a threshold ("free bits") and balanced between prior and posterior, and categorical distributions are mixed with 1% of a uniform distribution to avoid zero probabilities. With these changes, DreamerV3 was reported to outperform specialized methods across more than 150 tasks with a single configuration, and to be the first algorithm

to collect diamonds in Minecraft from scratch, without human data or curricula, a task that requires long sequences of actions from sparse rewards.

15.6.1 TD-MPC: planning in a latent space with a learned value ●●

TD-MPC (Hansen et al. 2022) combines the two uses of models that we have kept apart. It learns a latent model *without* reconstructing observations: an encoder, latent dynamics, a reward head, a terminal Q-function and a policy, trained jointly by predicting rewards, by temporal-difference learning of the Q-function, and by a *latent consistency* loss that makes the predicted next latent state match the encoding of the real next observation. It then acts by MPC in the latent space, with MPPI over a short horizon and the learned Q-function as terminal value, and it uses the learned policy to propose some of the candidate sequences. The short horizon keeps model errors small; the terminal value supplies the long-term view that pure MPC lacks. TD-MPC2 (Hansen et al. 2024) made the method robust across domains with one set of hyperparameters (normalizing latent states, using discrete regression for values) and reported that a single agent could be trained on 80 continuous-control tasks across several domains and embodiments, with performance improving as the model grew.

15.7 Learned models in games: AlphaZero and MuZero

Board games were the first great success of planning, and they show what a model can do when it is perfect. In chess or Go the rules are a perfect model, and the difficulty lies entirely in the size of the search tree. Monte Carlo tree search (MCTS, Algorithm 8.5) grows a search tree selectively, guided by statistics of simulated games (Section 8.5.3).

AlphaZero. Silver et al. (2018) combined MCTS with a single neural network $f_\theta(s) = (p, v)$ that outputs a probability distribution p over moves (the *prior*) and an estimate v of the game's outcome. In each search, the tree is descended by choosing at each node the move that maximizes

$$Q(s,a) + c\, p(a \mid s)\, \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)},$$

the average value $Q(s,a)$ of the move found so far, plus an exploration bonus that is large for moves the network likes and that have been tried rarely ($N(s,a)$ counts visits); a new leaf is evaluated by the network rather than by a random rollout. After the search, the visit counts at the root define an improved policy π_{MCTS}, which is sharper and better than p. The network is trained on games of *self-play* to match the search policy, $p \approx \pi_{\text{MCTS}}$, and the final outcome, $v \approx z$. Search acts as a policy improvement operator and the network as a policy evaluation and generalization device: a form of generalized policy iteration (Section 4.6). Starting from random play and given only the rules, AlphaZero was reported to defeat world-champion programs in chess, shogi and Go.

MuZero. In Atari games, robotics or anything outside board games, the rules are not given. MuZero (Schrittwieser et al. 2020) learned a model that is used only inside the search, made of three functions (Figure 15.6b): a representation function $s^0 = h_\theta(o_1, \dots, o_t)$ that encodes the

observations so far; a dynamics function $(r^k, s^k) = g_\theta(s^{k-1}, a^k)$ that predicts a reward and the next hidden state for a hypothetical action; and a prediction function $(p^k, v^k) = f_\theta(s^k)$ as in AlphaZero. MCTS runs entirely on hidden states. For a position at time t in the replay data, with the actions actually taken afterward, the model is unrolled K steps (five in the paper) and trained so that its predictions match three targets: the observed rewards u_{t+k}, value targets z_{t+k} (the game outcome in board games, an n-step bootstrapped return elsewhere), and the search policies π_{t+k} recorded when acting:

$$\ell(\theta) = \sum_{k=0}^{K} \Big[\ell^r\big(u_{t+k}, r^k\big) + \ell^v\big(z_{t+k}, v^k\big) + \ell^p\big(\pi_{t+k}, p^k\big) \Big] + c\,\|\theta\|^2,$$

with squared or cross-entropy losses (the reward term starts at $k = 1$). Nothing asks the hidden state to reconstruct an observation. MuZero was reported to match AlphaZero's performance in chess, shogi and Go without being told the rules, and to set a new state of the art on the 57 games of the Atari benchmark at the time. Later variants extended it to large or continuous action spaces (Hubert et al. 2021), to stochastic environments (Antonoglou et al. 2022) and to the data-limited Atari 100k setting (Ye et al. 2021).

HISTORICAL NOTE

Learning a model and planning with it is as old as adaptive control, where the certainty-equivalence principle (act as if the estimated model were true) appeared in the 1950s. In reinforcement learning, Sutton (1990) proposed Dyna, which integrated learning, planning and acting, and PILCO showed in 2011 that a probabilistic model could make learning on real hardware fast (Deisenroth and Rasmussen 2011). The deep era brought ensembles for uncertainty (PETS, MBPO), latent imagination (World Models, PlaNet, Dreamer) and learned models for search (MuZero).

15.7.1 Value equivalence ●●

MuZero's model is not a model of the environment in the usual sense: its hidden states need not correspond to anything observable. What, then, is it a model *of*? Grimm et al. (2020) gave an answer: it only needs to be *value equivalent* to the environment, that is, indistinguishable from it as far as the Bellman updates used for planning are concerned. Precursors include the predictron (Silver, Hasselt, et al. 2017) and value prediction networks (Oh et al. 2017), which also learned models end to end from value targets.

Definition 15.3 Value equivalence

Let Π be a set of policies and $\mathcal{V}$ a set of functions $\mathcal{S} \to \mathbb{R}$. A model $\tilde{M} = (\tilde{P}, \tilde{r})$ is *value equivalent* to the environment $M = (P, r)$ with respect to Π and $\mathcal{V}$ if their Bellman operators agree on these sets:

$$\tilde{\mathcal{T}}^\pi v = \mathcal{T}^\pi v \qquad \text{for all } \pi \in \Pi \text{ and } v \in \mathcal{V},$$

where $\mathcal{T}^\pi v = r^\pi + \gamma P^\pi v$ and $\tilde{\mathcal{T}}^\pi v = \tilde{r}^\pi + \gamma \tilde{P}^\pi v$.

Two facts make the definition useful. First, if $\tilde{M}$ is value equivalent to M with respect to Π and a set $\mathcal{V}$ that contains the true values V_M^π of the policies in Π, then $\tilde{M}$ evaluates every $\pi \in \Pi$

exactly. Indeed $\tilde{\mathcal{T}}^\pi V_M^\pi = \mathcal{T}^\pi V_M^\pi = V_M^\pi$, so V_M^π is a fixed point of $\tilde{\mathcal{T}}^\pi$, which is a γ-contraction (Theorem 4.3) and has only one fixed point, $V_{\tilde{M}}^\pi$. Second, the larger Π and $\mathcal{V}$, the fewer models qualify; if Π contains all policies and $\mathcal{V}$ all functions, only the true model remains. In between lies a trade-off: a value-equivalent model for a small set of policies and functions can be much simpler than the true model, because it may ignore every aspect of the environment that does not affect those values, which is the point when the environment is an image stream and the decisions depend on a few objects in it. Exercise 15.15 constructs such a model for a small MDP. MuZero's loss can be read as asking for value equivalence along the trajectories of the search policy.

15.8 How much data does a model need? ●●●

The simulation lemma suggests an approach to the sample complexity of model-based RL: estimate every transition distribution to L^1 accuracy ϵ_P and plan. With a *generative model*, a simulator that returns a sample $s' \sim P(\cdot \mid s, a)$ for any queried pair, estimating one distribution over $|\mathcal{S}|$ states to L^1 accuracy ϵ_P takes on the order of $|\mathcal{S}|/\epsilon_P^2$ samples, and Corollary 15.2 then needs $\epsilon_P \approx \epsilon(1-\gamma)^2/r_{\max}$ for an ϵ-optimal policy. Ignoring constants and logarithms, this route costs about $|\mathcal{S}|^2|\mathcal{A}|/((1-\gamma)^4\epsilon^2)$ samples.

This is far from the truth. Azar et al. (2013) showed that with N samples per pair, the values of the estimated model satisfy $\|\hat{Q}^* - Q^*\|_\infty \le \epsilon$ with high probability once the total number of samples is $\tilde{O}\big(|\mathcal{S}||\mathcal{A}|/((1-\gamma)^3\epsilon^2)\big)$ (for ϵ small enough), where $\tilde{O}$ hides constants and logarithmic factors, and that no method can estimate Q^* with fewer samples in the worst case. Agarwal, Kakade, et al. (2020) proved that the *plug-in policy* itself, the certainty-equivalence policy of Section 15.1.1, is ϵ-optimal with the same order of samples for ϵ up to about $1/\sqrt{1-\gamma}$, and Li et al. (2020a) extended the guarantee to the full range of ϵ by planning in a slightly perturbed model. So the simplest model-based method is minimax optimal in this setting: the number of samples per state–action pair does not grow with $|\mathcal{S}|$, although the model has $|\mathcal{S}|$ parameters per pair.

Why does the plug-in approach beat the simulation-lemma route by a factor of $|\mathcal{S}|/(1-\gamma)$? The identity (15.2) shows that only the errors $(\hat{P} - P)V$ matter, one number per pair, not the full L^1 error of the distribution. For a fixed vector V, $(\hat{P}(\cdot \mid s, a) - P(\cdot \mid s, a))V$ is an average of N independent bounded terms, so by Hoeffding's or Bernstein's inequality (Theorem B.9 and Theorem B.11) it is of order $\sqrt{\mathrm{Var}(V)/N}$, with no dependence on $|\mathcal{S}|$. The difficulty is that the relevant V (the value of the plug-in policy, or $\hat{V}^*$) depends on the same samples; the cited proofs break this dependence with clever arguments (for example, by comparing with auxiliary MDPs in which one state is made absorbing) and use variance bounds, on the variance of the return, to save a factor of $1/(1-\gamma)$.

Without a generative model the agent must also *explore*: it can only sample transitions from states it reaches. Model-based ideas dominate the theory of this setting too. E^3 (Kearns and Singh 2002) and R-max (Algorithm 16.1) build models of the "known" part of the MDP and are optimistic about the rest; UCRL2 (Jaksch et al. 2010) plans optimistically over all models consistent with confidence intervals; posterior sampling (PSRL, Algorithm 16.2; Osband et al.

(2013)) plans in a model sampled from the posterior. Chapter 16 develops these methods and their regret bounds. With function approximation the picture is much less complete, and the role of models (versus value-based methods) in large problems is an active research question.

RESEARCH NOTE

Open questions: When does a learned parametric model help more than replay or a high update-to-data ratio, and can we predict it from properties of the task? How should a model be trained when its only purpose is planning, so that it is accurate where the planner looks (objective mismatch, value equivalence)? How should model uncertainty be used: pessimism for safety and offline RL (Chapter 17), optimism for exploration (Chapter 16), or both? And what is the right way to combine decision-time planning with learned policies at the scale of foundation models? Hamrick et al. (2021) and Moerland et al. (2023) are good starting points.

15.9 Summary

- A model predicts how the environment responds to actions. Learned models can make agents far more sample efficient, allow lookahead at decision time and transfer to new goals, but they are wrong, their errors compound over long predictions, and planners exploit their mistakes.
- Models enter agents in four ways: generating experience for a model-free learner (Dyna, MBPO), planning at decision time (MPC, MCTS), improving learning targets (MVE, STEVE), and as differentiable simulators (PILCO, Dreamer).
- Certainty equivalence estimates a tabular model by counting and plans in it. On the slippery Trail it finds the optimal policy from 26 transitions; untried actions need optimism, or the agent may never try them.
- Learned dynamics models should predict small, standardized changes in a well-chosen representation. Ensembles estimate epistemic uncertainty, probabilistic outputs capture aleatoric noise, and models must be judged by their multi-step errors on the data the planner will see.
- (●●) The simulation lemma bounds the value error of every policy by $\epsilon_r/(1-\gamma) + \gamma\epsilon_P r_{\max}/(1-\gamma)^2$, and planning in the model loses at most twice that. The bound is tight in general but loose on typical problems.
- Model-predictive control optimizes short action sequences in the model at every step and replans from the real state; CEM, random shooting and MPPI are the usual optimizers. On the Pendulum, CEM-MPC with a model learned from 400 random steps swings up as well as with the exact simulator.
- MBPO trains an off-policy learner on many short model rollouts branched from real states. Much of its gain comes from doing more updates per real step, and on small problems replay with a high update ratio can do as well.
- Latent world models (World Models, PlaNet, Dreamer, TD-MPC) learn compact states from images and plan or learn behavior in imagination; DreamerV3 works across domains with fixed hyperparameters thanks to scale-robust losses.
- AlphaZero combines MCTS with a policy-value network; MuZero learns a model used only inside the search, a value-equivalent model trained to predict rewards, values and search

policies.

- (●●●) With a generative model, certainty equivalence is minimax optimal, needing $\tilde{O}(|\mathcal{S}||\mathcal{A}|/((1-\gamma)^3\epsilon^2))$ samples.

The next chapter takes up the question this chapter kept postponing: how an agent should collect the data from which it learns its model and its values.

15.10 Further Reading

The survey by Moerland et al. (2023) is the most complete map of model-based RL, organized by how the model is learned and how planning is integrated with learning; read it after this chapter. Chapter 8 of Sutton and Barto (2018) covers the tabular foundations (Dyna, prioritized sweeping, decision-time planning) from which this chapter starts.

For models and their uncertainty, read PILCO (Deisenroth and Rasmussen 2011) for the case for probabilistic models, then PETS (Chua et al. 2018), which is short, clear and full of useful ablations about ensembles and trajectory sampling. Lambert et al. (2020) is a thought-provoking analysis of why likelihood is the wrong objective, and Farahmand et al. (2017) introduced value-aware model learning.

For learning policies from models, MBPO (Janner et al. 2019) contains both the algorithm and its theory; read it together with the replies of Hasselt et al. (2019) and X. Chen et al. (2021), which ask how much of the gain is due to the model. MVE (Feinberg et al. 2018) and STEVE (Buckman et al. 2018) are short papers on model-based targets. For MPC, Williams et al. (2017) present MPPI from first principles, and the tutorial of de Boer et al. (2005) covers the cross-entropy method in general.

For world models, the interactive article version of Ha and Schmidhuber (2018) is a delight; PlaNet (Hafner et al. 2019) introduced the RSSM; and the DreamerV3 paper (Hafner et al. 2025) is worth reading for its robustness techniques alone. TD-MPC (Hansen et al. 2022) and TD-MPC2 (Hansen et al. 2024) show the alternative without reconstruction. For games, read AlphaZero (Silver et al. 2018) and MuZero (Schrittwieser et al. 2020) in that order, then the value-equivalence paper of Grimm et al. (2020) for the theory behind MuZero's kind of model.

For theory, Azar et al. (2013) and Agarwal, Kakade, et al. (2020) are the key papers on certainty equivalence with a generative model, and Li et al. (2020a) closes the remaining gap; the book draft of Agarwal et al. (2022) presents the generative-model analysis in a textbook form. For the role of planning in deep model-based RL, the empirical study of Hamrick et al. (2021) is a good entry point to the research questions.

15.11 Exercises

Exercise 15.1 ● Certainty equivalence on the Trail

An agent collects data on the slippery Trail ($\gamma = 0.9$, true slip probability 0.2): exit at Start twice (reward 1); go at Start 8 times, reaching Middle 6 times and staying twice; exit at Middle twice

(reward 2); go at Middle 10 times, reaching Goal 9 times and staying once; and finish 5 times (reward 4). (a) Write down the maximum-likelihood model. (b) Compute $\hat{V}$ and the plug-in policy. (c) Compare with $V^*(\text{Start}) = 3.0839$ and $V^*(\text{Middle}) = 3.5122$; at which state is the value error larger, and why is the error at Start smaller than you might expect?

Exercise 15.2 ●● The simulation lemma is tight

Consider an MDP with one action, a state s_0 with reward 1 that stays in s_0 forever, and an absorbing state with reward 0. A model says that from s_0 the agent moves to the absorbing state with probability $\epsilon/2$ per step (and stays with probability $1-\epsilon/2$). (a) Compute ϵ_P, ϵ_r and $r_{\max}$. (b) Compute the true and the model value of s_0 and their difference. (c) Show that the difference is at least a constant fraction of the bound of Lemma 15.1 when ϵ is small compared with $1-\gamma$, so that the $1/(1-\gamma)^2$ factor cannot be improved in general. Give numbers for $\gamma = 0.9$, $\epsilon = 0.02$.

Exercise 15.3 ●● A sharper simulation lemma

Suppose all rewards lie in $[0, r_{\max}]$. Show that the transition term of Lemma 15.1 can be halved: $\|V^\pi_{\hat{M}} - V^\pi_M\|_\infty \le \epsilon_r/(1-\gamma) + \gamma\,\epsilon_P\, r_{\max}/(2(1-\gamma)^2)$. Where exactly does the proof use that $\hat{P}(\cdot \mid s,a)$ and $P(\cdot \mid s,a)$ both sum to one?

Exercise 15.4 ● Compounding error of a linear model

A one-dimensional system evolves as $x_{t+1} = a\,x_t$ from $x_0 = 1$, and a learned model uses $\hat{a} = a + \delta$. (a) Write the error $\hat{x}_k - x_k$ after k open-loop steps and its first-order approximation in δ. (b) For $a = 0.9$ and $\delta = 0.05$, compute the error for $k = 1, 5, 10, 20, 50$; at which k is it largest? (c) Repeat for $a = 1.1$. What does the comparison say about learning models of stable and unstable systems?

Exercise 15.5 ●● Exponential compounding

Let the true dynamics be deterministic, $s_{t+1} = f(s_t, a_t)$, with $\|f(x,a) - f(y,a)\| \le L\|x - y\|$ for all x, y, a, and let a model $\hat{f}$ satisfy $\|\hat{f}(x,a) - f(x,a)\| \le \epsilon$ everywhere. Show that the open-loop prediction error after k steps is at most $\epsilon(L^k - 1)/(L-1)$ (or $k\epsilon$ if $L = 1$), and give an example in which this bound is attained. Compare with (15.3).

Exercise 15.6 ● How should a model be evaluated?

A colleague reports that their learned dynamics model has a test mean-squared error of 10^{-4} and asks whether it is good enough for planning. List the questions you would ask before answering, and describe an evaluation protocol of three or four measurements that would convince you. Use Figure 15.3 to explain why each measurement matters.

Exercise 15.7 ● Why MBPO uses short rollouts

Explain in your own words, using Section 15.2 and the simulation lemma stated in Section 15.1.1, why MBPO starts its model rollouts from states of the real replay buffer and stops them after a few steps. What would go wrong with rollouts of length 0, and with rollouts of the full episode length from the initial state? Why might the best rollout length grow during training?

Exercise 15.8 • Reading an ensemble

At some input, three probabilistic ensemble members predict Gaussians with means 1.0, 1.4 and 0.6 and variances 0.04, 0.09 and 0.05. (a) Compute the aleatoric, epistemic and total variance. (b) Should a planner trust this prediction? Would more data at this input help? (c) Describe a situation in which all members agree but are all wrong, and explain why ensembles cannot detect it.

Exercise 15.9 •• Where the cross-entropy method gets its name

Let p be a distribution over action sequences and $\mathcal{E}$ the event that a sequence is among the elites (its predicted return exceeds a threshold). (a) Show that the Gaussian $q = \mathcal{N}(m, \operatorname{diag}\sigma^2)$ that minimizes the cross-entropy $-\mathbb{E}_{x\sim p(\cdot\mid\mathcal{E})}[\log q(x)]$ has m and σ^2 equal to the mean and variance of $p(\cdot \mid \mathcal{E})$. (b) Conclude that the CEM update in Algorithm 15.1 is the sample version of this minimization. (c) Why does CEM usually need a minimum standard deviation or smoothing in practice?

Exercise 15.10 •• MPPI weights from a KL-regularized problem

Given N sampled sequences with costs $C_1, \dots, C_N$, find the weights $w \in \Delta_N$ (the probability simplex) that minimize $\sum_i w_i C_i + \lambda \sum_i w_i \log(N w_i)$, that is, the expected cost plus λ times the KL divergence from the uniform distribution. Show that the solution is the MPPI weighting (15.4), and describe the limits $\lambda \to 0$ and $\lambda \to \infty$.

Exercise 15.11 •• Bias of the expanded target

Assume a deterministic environment, a perfect model, and a critic with error $|\bar{Q}(s,a) - Q^\pi(s,a)| \le \epsilon_Q$ everywhere. (a) Show that the error of the MVE target (15.5) as an estimate of $Q^\pi(s_t, a_t)$ is at most $\gamma^{H+1}\epsilon_Q$. (b) Now let the model be imperfect: each imagined reward has error at most ϵ_r, and because the last imagined state is wrong, the critic's value there has error at most $\epsilon_Q + \epsilon_s$. Bound the target error and find the horizon H that minimizes your bound. (c) What does this say about choosing H, and what changes if the reward error grows with the number of imagined steps?

Exercise 15.12 •• Counting samples

Take $|\mathcal{S}| = 100$, $|\mathcal{A}| = 4$, $\epsilon = 0.1$ and rewards in $[0,1]$. Compare, ignoring constants and logarithms, the number of samples per state–action pair suggested by (a) the simulation-lemma route of Section 15.8 and (b) the minimax rate $1/((1-\gamma)^3\epsilon^2)$, for $\gamma = 0.9$ and $\gamma = 0.99$. Which factor accounts for most of the difference, and why does the plug-in approach not pay it?

Exercise 15.13 ● Programming: the planning horizon

Extend Task 4 of Lab 15.1: from the hanging position, run CEM, MPPI and random shooting with horizons $H = 10, 15, 20, 30$, keeping the number of samples per step fixed. Report the returns. Which planner degrades with longer horizons, and why? What happens to MPPI with the shortest horizon?

Exercise 15.14 ● Programming: rollout length in MBPO

Run the MBPO-style learner of Lab 15.1 with rollout lengths $k = 1, 3, 10$ for 12 episodes and three seeds. Does the rollout length matter on the Pendulum? Relate your answer to the open-loop errors you measured in Task 3, and predict what would happen with a model trained on much less data.

Exercise 15.15 ●●● A value-equivalent model that is not the true model

Consider the slippery Trail with $\gamma = 0.9$ and the policy π that always goes. (a) Find a model $\tilde{M}$ that differs from the true model at (Start, go) (it never slips there, but it gives a reward, possibly negative, for moving on) and that is value equivalent to the true model with respect to $\Pi = \{\pi\}$ and $\mathcal{V} = \{V^\pi\}$. (b) Verify that $\tilde{M}$ gives the correct value of π at every state. (c) Show that $\tilde{M}$ gives the wrong value for the policy that goes at Start and exits at Middle. (d) Relate this to MuZero, whose model is trained only along the trajectories of its search policy.

Exercise 15.16 ●●● When do learned models beat model-free methods?

Write a two-page research memo on the question in the title. Use the evidence in this chapter (Figure 15.5), Hasselt et al. (2019), X. Chen et al. (2021), Hamrick et al. (2021) and the successes of MuZero and DreamerV3. Identify at least three properties of a task or a setting that favor learned models, propose one small, decisive experiment for each, and say what result would change your mind.

CHAPTER 16

Exploration

WHAT YOU WILL LEARN

- Explain why exploration is harder in sequential problems than in bandits, and why dithering methods such as ε-greedy can need exponentially many episodes.
- Apply optimism in the face of uncertainty to MDPs: optimistic initialization, R-max and count-based exploration bonuses.
- Describe posterior sampling for reinforcement learning (PSRL) and its approximations by randomized value functions and bootstrapped ensembles.
- Use intrinsic rewards for deep reinforcement learning (pseudo-counts, curiosity, random network distillation) and recognize their failure modes, such as the noisy-TV problem.
- Diagnose exploration problems in practice and choose a suitable method.
- Define PAC-MDP sample complexity and regret, and sketch the "optimism plus sum of bonuses" proof of a regret bound for UCBVI (●●).
- Survey exploration with function approximation: linear MDPs, structural complexity measures and lower bounds (●●●).

Chapter 2 studied exploration in its simplest form: one situation, several actions, unknown rewards. The lessons were clear. An agent that always acts greedily can lock onto a poor action forever. Random exploration (ε-greedy) fixes this but wastes effort, while directed methods, such as upper confidence bounds and Thompson sampling, explore where it matters and achieve regret that grows only logarithmically.

Sequential problems add a new difficulty. In an MDP the information an agent needs may be far away: to learn what lies at the end of a corridor, it must first walk down the corridor, which may take many deliberate steps that look pointless until the end is reached. On the Trail of Chapter 1, an agent that does not know about the +4 at Goal must choose go twice in a row before it can see it; on a trail with twenty stations it would need twenty. Exploring one step at a time, by coin flips, finds such rewards only by luck, and the luck required grows exponentially with the distance. Agents need *deep exploration*: plans that deliberately visit unknown parts of the world.

This chapter develops the two principles of Chapter 2 into methods for MDPs. *Optimism* (Section 16.2) treats what the agent has not tried as if it were good, and lets planning carry that optimism back to the start. *Posterior sampling* (Section 16.3) commits for a whole episode to one plausible guess about the world. Section 16.1 first shows why exploring step by step fails. Section 16.4 and 16.5 (●●) define how exploration is measured and prove what optimism achieves; Section 16.6 (●●●) surveys what is known with function approximation. Section 16.7 turns to deep reinforcement learning, where counts are replaced by intrinsic rewards, and Section 16.8

gives practical advice. On a first reading, take the ● sections and the lab.

16.1 Why exploration is hard in MDPs

In a bandit, every pull of every arm is equally informative and equally available: the agent can try any arm in any round. In an MDP, the agent can try an action only in a state it has reached, and reaching some states takes a long sequence of the right actions. The value of exploring an action therefore depends on everything the agent would have to do to make use of what it learns. Exploration has to look ahead.

16.1.1 Dithering

The simplest exploration strategies of Chapter 2 carry over to MDPs unchanged. An ε-*greedy* agent takes a random action with probability ε in every state and the greedy action otherwise. A *Boltzmann* or *softmax* agent takes action a in state s with probability proportional to $e^{Q(s,a)/\tau}$, where the temperature $\tau > 0$ controls how random it is. Both perturb each decision independently: they *dither* around the greedy policy. In many problems that is enough, and ε-greedy remains the default exploration of deep Q-learning (Chapter 10). It fails when a reward can only be reached by a long sequence of specific actions, because independent random perturbations rarely produce long specific sequences.

Example 16.1 A long Trail

Stretch the Trail to N stations. At each station the hiker can exit, collecting a small reward, or go on; only the last station holds the big reward. A hiker who does not yet know where the big reward is and explores by flipping a fair coin at every station reaches the last station with probability $(\frac{1}{2})^{N-1}$ per episode. On the Trail itself ($N = 3$) that is $\frac{1}{4}$, so the +4 is found after four episodes on average. With $N = 11$ stations it takes $2^{10} = 1024$ episodes on average, and with $N = 21$ about a million.

♦

Coin flipping is the most generous version of dithering. An ε-greedy agent that has already formed the belief that exiting is best (because exiting pays something and going on has paid nothing so far) goes on only on its random moves, with probability $\varepsilon/2$ at each station, and needs about $(2/\varepsilon)^{N-1}$ episodes: with $\varepsilon = 0.1$ and $N = 6$, more than three million.

16.1.2 The deep sea

A standard test of deep exploration is the *deep sea* of Osband et al. (2019), shown in the left panel of Figure 16.1. The agent starts at the top-left cell of an $N \times N$ grid and descends one row per step, moving one column to the left or to the right as it chooses. Moving right costs a little, $0.01/N$ per move; moving right in the bottom-right cell earns +1. The episode lasts N steps, so the only rewarding behavior is to go right N times in a row, for a total of $1 - 0.01 = 0.99$; every other behavior earns at most 0. A uniformly random policy finds the treasure with probability

2^{-N}, and the small cost of moving right actively discourages an agent from trying: until it finds the treasure, "left" looks better in every state.

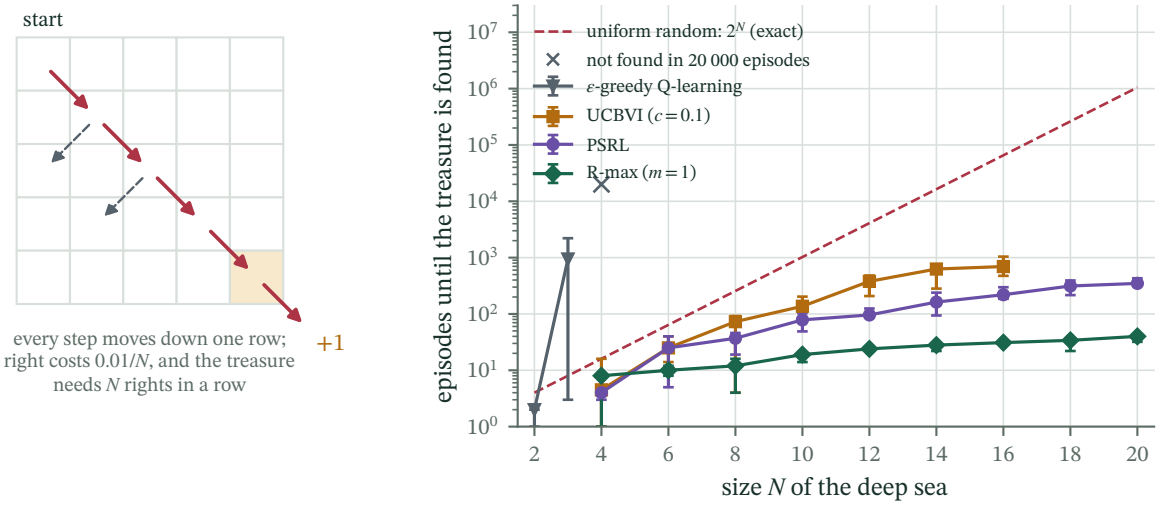


Figure 16.1. The deep sea. Left: an $N \times N$ grid ($N = 5$); the agent moves down one row per step and chooses left or right; the treasure (amber) is reached only by going right N times in a row (red). Right: the number of episodes until the treasure is first found, for several sizes N (median of 4 runs, bars from the smallest to the largest). Dithering is exponential: a random policy needs 2^N episodes on average, and ε-greedy Q-learning ($\varepsilon = 0.1$) already fails within 20 000 episodes at $N = 4$ in some runs. The directed methods of this chapter grow polynomially: R-max ($m = 1$) needs about $2N$ episodes, PSRL a few hundred at $N = 20$, UCBVI with a small bonus somewhat more.

The right panel of Figure 16.1 measures how long various agents need to find the treasure. The dashed line is the exact 2^N of random exploration. ε-greedy Q-learning is no better than its exponential bound suggests: it finds the treasure quickly for $N = 2$, often needs hundreds or thousands of episodes for $N = 3$, and fails within 20 000 episodes in some runs at $N = 4$. The other three methods, introduced in the next two sections, find the treasure after a number of episodes that grows only polynomially with N: about $2N$ for R-max, a few hundred episodes at $N = 20$ for posterior sampling.

KEY IDEA

Exploration in an MDP must be *deep*: an agent must be willing to follow a long sequence of actions whose payoff is uncertain and will only be learned at the end. Dithering perturbs each decision independently and therefore explores deep regions with a probability that decays exponentially with their distance. Directed methods plan their exploration.

Two remarks prevent misunderstandings. First, the deep sea is not an exotic construction. Long corridors, keys that must be fetched before a door can be opened, and molecules that must be assembled in many steps before they show any useful property are all common, and the hard-exploration Atari games of Section 16.7 are of this kind. Second, dithering is not useless. When rewards are dense and the reward of a good action shows up quickly, ε-greedy and entropy

bonuses work well, and they are cheap. The trouble starts when the useful information is many deliberate steps away.

16.2 Optimism in the face of uncertainty

Chapter 2 introduced the principle of *optimism in the face of uncertainty*: act as if every uncertain option were as good as it could plausibly be. If the optimism is justified, the agent exploits a good option; if not, it learns, and its optimism about that option fades. In an MDP the principle becomes more effective than in a bandit, because the agent plans: optimism about an untried action deep in the state space raises the value of every state from which that action can be reached, and planning turns this into a path that leads the agent there.

16.2.1 Optimistic initialization revisited

The simplest optimistic method starts every value estimate high. If rewards lie in $[0, r_{\max}]$, no discounted return exceeds $r_{\max}/(1-\gamma)$, and an episode of H steps cannot earn more than $Hr_{\max}$. Initializing $Q(s,a)$ at that maximum and acting greedily makes every untried action look attractive. In a bandit, the effect lasts only until each arm is tried; in an MDP, it spreads. A temporal-difference update (Chapter 6) such as Q-learning's moves $Q(s,a)$ toward $r + \gamma \max_{a'} Q(s',a')$. As long as s' has an untried action, this target stays high, so the agent keeps returning to s and moving on to s', toward the unexplored action. Optimism flows backward from the frontier of the explored region, one update at a time.

Optimistic initialization is a useful default for small problems, but it has two weaknesses. It forgets its optimism as soon as an action has been tried a few times, even if a few tries are far from enough to know its value; with stochastic rewards or transitions, an unlucky first experience can make a good action look bad for a long time. And the optimism travels backward only as fast as the agent's updates carry it, which can be slow. The methods below fix both: they keep track of how much the agent knows about each action, and they plan with a model so that optimism spreads immediately.

16.2.2 R-max and the known-state idea

R-max, by Brafman and Tennenholtz (2002), makes the idea of "how much the agent knows" explicit. It counts the visits $N(s,a)$ to every state–action pair and calls a pair *known* once $N(s,a) \geq m$, for a threshold m. For known pairs it trusts the empirical model, the observed frequencies of next states and the average of the observed rewards. For unknown pairs it assumes the best possible outcome: the pair pays the maximal reward $r_{\max}$ at every remaining step. It then computes the optimal policy of this optimistic model and follows it.

The result is an agent that heads straight for the nearest unknown pair, if reaching it is worth the trip, and exploits its knowledge otherwise. Either it gets close to the optimal value of the true MDP, or it reaches unknown pairs and learns about them; since each pair can become known only once, the second case can happen only a bounded number of times. This is the *known-state* argument of Kearns and Singh (2002), whose E^3 algorithm was the first with a polynomial guarantee, and R-max is its simplest form.

Algorithm 16.1 R-max (episodic version)

Input: horizon H, largest reward $r_{\max}$, known-ness threshold m.

1. Set $N(s,a) \leftarrow 0$ and clear the transition and reward statistics of every pair.
2. For each episode:
   1. Let $\hat{P}$ and $\hat{r}$ be the empirical transition probabilities and mean rewards of the known pairs ($N(s,a) \geq m$).
   2. For $h = H-1, \dots, 0$ and all s, a: $Q_h(s,a) \leftarrow \hat{r}(s,a) + \sum_{s'} \hat{P}(s' \mid s,a) \max_{a'} Q_{h+1}(s',a')$ if (s,a) is known, and $Q_h(s,a) \leftarrow (H-h)\, r_{\max}$ otherwise (with $Q_H = 0$).
   3. Follow the greedy policy $a_h = \arg\max_a Q_h(s_h, a)$ for H steps, updating the counts and statistics of the pairs visited.

Output: the final greedy policy.

Here Q_h is the value with $H-h$ steps to go; the time index is needed because a finite-horizon problem's optimal policy can depend on how much time is left (Chapter 3). The algorithm replans before every episode; since the model changes only when a pair becomes known, replanning at those moments is enough. With deterministic dynamics, one visit tells the agent everything about a pair, and $m = 1$ suffices. With stochastic dynamics, m must be large enough for the empirical model of a known pair to be accurate; the analysis takes m of order $H^4(S + \ln(SA/\delta))/\varepsilon^2$ (with $1/(1-\gamma)$ in place of H in the discounted setting), by combining a concentration inequality with the simulation lemma of Lemma 15.1 (Exercise 16.5).

Example 16.2 R-max on the Trail

Run R-max on the Trail without discounting ($H = 3$ steps, $r_{\max} = 4$, $m = 1$), breaking ties toward exit. In episode 1 nothing is known, so every action at Start is worth $3 \times 4 = 12$; the tie goes to exit, which pays 1 and becomes known. In episode 2, exit at Start is worth 1 while the unknown go is still worth 12, so the hiker goes on; at Middle both actions are unknown and worth $2 \times 4 = 8$, and the tie goes to exit, which pays 2. In episode 3, go at Start is known to lead to Middle, whose unknown go is still worth 8, so the hiker goes on twice and collects the +4 at Goal. From then on everything relevant is known, and the hiker goes all the way in every episode. The policies of the three episodes earned 1, 2 and 4: three episodes of exploration, no more. On a Trail with N stations the same reasoning takes N episodes, against about 2^{N-1} for coin flipping. The lab's tests check both counts.

♦

16.2.3 Exploration bonuses

R-max's optimism is all or nothing: a pair is either unknown and maximally attractive, or known and taken at face value. A smoother alternative keeps every estimate but adds a bonus that

shrinks with the number of visits, as UCB1 did in Chapter 2. The *exploration bonus*

$$b(s,a) = \frac{\beta}{\sqrt{N(s,a)}}$$

is added to the reward of the pair before planning, or before a temporal-difference update. The scale β trades exploration against exploitation. The $1/\sqrt{N}$ shape comes from concentration: by Hoeffding's inequality (Theorem B.9), the average of N samples of a quantity with range R is within $R\sqrt{\ln(2/\delta)/(2N)}$ of its mean with probability at least $1-\delta$. A bonus of that size makes the optimistic estimate an upper bound on the truth with high probability.

The model-based version of this idea is MBIE-EB of Strehl and Littman (2008) (model-based interval estimation with exploration bonus). It solves the empirical model with the reward $\hat{r}(s,a) + \beta/\sqrt{N(s,a)}$ and acts greedily; with β of order $\frac{1}{1-\gamma}\sqrt{\ln(SA/\delta)}$ it has a polynomial guarantee in the discounted setting. In the finite-horizon setting, the same construction with a bonus of order $H\sqrt{\ln(SAT/\delta)/N(s,a)}$ is UCBVI, analyzed in Section 16.5. The model-free version adds the bonus inside Q-learning; with suitable step sizes this also has guarantees (Section 16.5), and with counts replaced by learned density estimates it becomes the count-based exploration of deep reinforcement learning (Section 16.7).

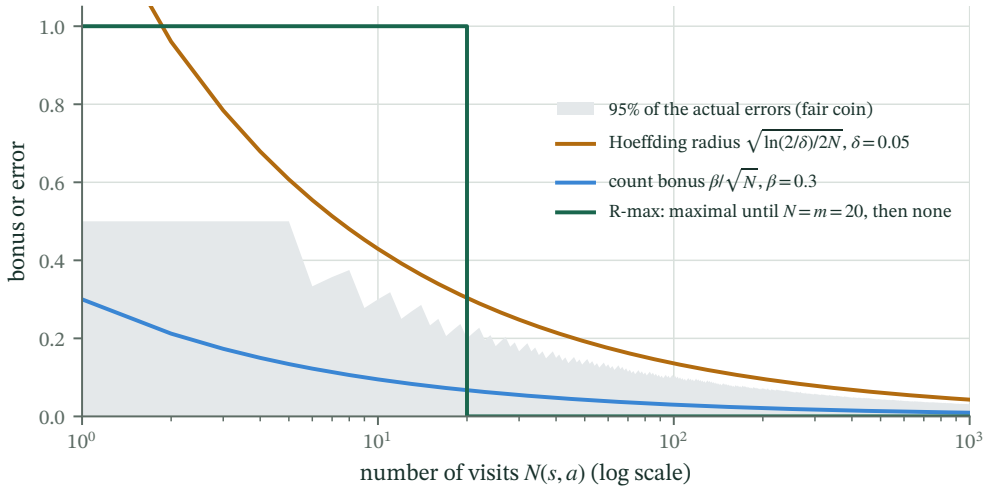


Figure 16.2. Two ways to be optimistic. R-max (green) treats a pair as maximally valuable until it has been visited m times and then trusts its estimate completely. A count bonus $\beta/\sqrt{N}$ (blue) shrinks gradually. The Hoeffding radius (amber), the bonus that theory prescribes, lies above 95% of the actual errors of an average of N fair coin flips (gray band), as it must, but by a wide margin.

Figure 16.2 compares the shapes. The theoretical bonus is safe but conservative: it must hold for every distribution with the given range and every one of many estimates at once, so in a particular problem it is usually much larger than the actual error. This matters in practice. On RiverSwim, below, a bonus ten times smaller than the theoretical one learns far faster, and one a hundred times smaller faster still. But a bonus of zero, which leaves only R-max-style optimism about pairs never tried, fails there: after one try, the noisy estimate of the rewarding action usually looks bad, and the agent never returns to it. Some optimism must survive the first few visits.

PITFALL

A count-based bonus is only as good as the counts. In a tabular problem, $N(s, a)$ is unambiguous. With continuous or high-dimensional states, no state is ever visited twice, every raw count is 0 or 1, and the bonus carries no information. The counts must come from a notion of similarity between states, which is exactly what Section 16.7 is about.

16.2.4 RiverSwim

A second classic test problem, *RiverSwim* (Strehl and Littman 2008), shows how these methods behave when the dynamics are random. A swimmer is in a river with six positions. Swimming left, with the current, always succeeds, and at the left bank it earns a tiny reward of 0.005. Swimming right, against the current, succeeds only with probability 0.35 (it leaves the swimmer in place with probability 0.6 and drifts her back with probability 0.05), but at the right bank it earns a reward of 1 at every step. We use episodes of $H = 20$ steps starting at the left bank. Always swimming left earns 0.1 per episode. Always swimming right earns 3.063, and the optimal policy, which swims right except when too little time remains to reach the right bank, earns 3.064. An agent that judges by its first experiences sees that swimming right usually achieves nothing and settles for the sure tiny reward.

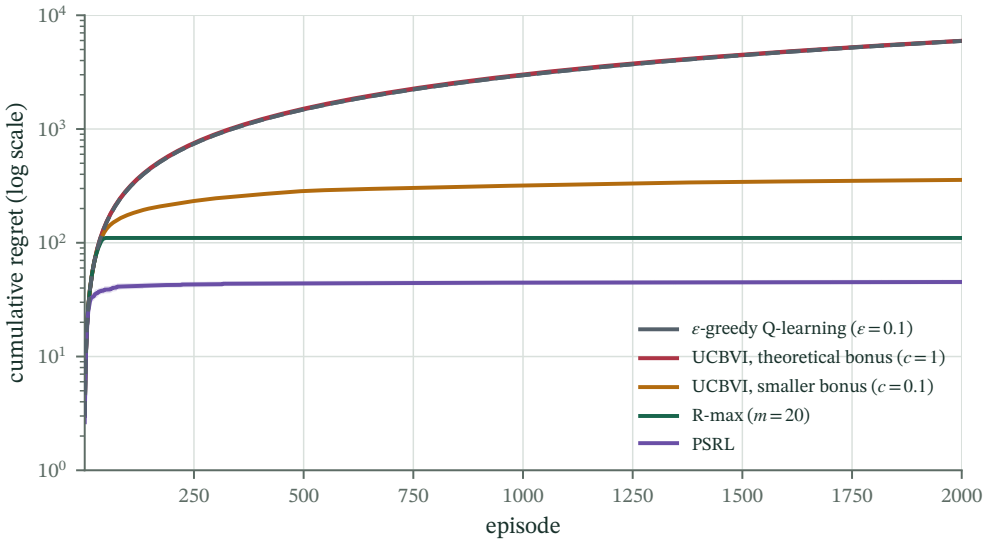


Figure 16.3. Cumulative regret on RiverSwim over 2000 episodes of 20 steps, averaged over 8 runs (bands: ±1 standard error; log scale). ε-greedy Q-learning (dashed) never discovers the right bank, and UCBVI with the bonus prescribed by theory ($c = 1$) is still exploring; both have regret growing linearly over this horizon. UCBVI with a ten times smaller bonus, R-max ($m = 20$) and PSRL all learn; PSRL has the smallest regret.

Figure 16.3 shows the regret, the total shortfall from the optimal value 3.064 per episode, accumulated over 2000 episodes. ε-greedy Q-learning has regret close to 3 per episode throughout: it rarely makes it to the right bank. UCBVI with the theoretical bonus ($c = 1$, in the notation of Algorithm 16.3) is also linear over this horizon, for the opposite reason: with $H = 20$ its bonus after 100 visits of a pair is about 6.2, twice the whole value of the optimal policy, so it keeps re-

exploring pairs it already knows well. Ten times smaller bonuses ($c = 0.1$) give a regret of about 357 after 2000 episodes, R-max with $m = 20$ about 110, and PSRL, the subject of the next section, about 45.

CHECKPOINT

1. Why does optimistic initialization explore more effectively in an MDP than in a bandit?
2. In R-max, what happens to the value of a state from which an unknown pair can be reached?
3. Why can a bonus of zero fail on RiverSwim even though R-max-style optimism about untried pairs remains?

Answers.

1. Because planning or bootstrapping propagates the optimism: states that lead toward untried actions inherit high values, so the agent is drawn along a path to them, not just toward untried actions in its current state.
2. It is raised: the unknown pair is assumed to pay $r_{\max}$ at every remaining step, and that value flows back through the planning step to every state from which the pair is reachable.
3. Because after one try the pair counts as tried, and a single noisy outcome of "swim right" usually looks worthless; with no bonus left, the agent stops trying it and settles for the small sure reward.

16.3 Posterior sampling

Thompson sampling (Section 2.5) explores a bandit by drawing one plausible mean for every arm from its posterior and playing the arm whose draw is best. It extends to MDPs in an elegant way: draw one plausible *MDP* from the posterior, and act optimally in it.

16.3.1 PSRL

The agent keeps a posterior distribution over the unknown transition probabilities and rewards. For finite MDPs a convenient choice is a *Dirichlet* distribution for each row $P(\cdot \mid s, a)$: the Dirichlet distribution with parameters $(\alpha_1, \dots, \alpha_S)$ is a distribution over probability vectors with mean $\alpha_i / \sum_j \alpha_j$, and it generalizes the Beta distribution of Chapter 2 to more than two outcomes. It is conjugate to observed transitions: starting from a prior Dirichlet$(\alpha_0, \dots, \alpha_0)$, the posterior after observing the next states of (s, a) is Dirichlet$(\alpha_0 + C(s, a, 1), \dots, \alpha_0 + C(s, a, S))$, where $C(s, a, s')$ counts the observed transitions from (s, a) to s'. The rewards get a conjugate posterior of their own, for example a Normal one.

Algorithm 16.2 Posterior sampling for reinforcement learning (PSRL)

Input: a prior over MDPs (for example Dirichlet rows for P and Normal priors for the mean rewards), horizon H.

1. For each episode $k = 1, 2, \dots$:
   1. Draw one MDP $\tilde{M}_k = (\tilde{P}_k, \tilde{r}_k)$ from the posterior given all data so far.
   2. Compute the optimal policy $\tilde{\pi}_k$ of $\tilde{M}_k$ by backward induction.

3. Follow $\tilde{\pi}_k$ for the whole episode, recording every transition (s_h, a_h, r_h, s_{h+1}).
4. Update the posterior with the episode's transitions (for Dirichlet rows: add one to the count of each observed next state).

Output: the posterior, and the policies $\tilde{\pi}_k$.

The idea goes back to Strens (2000); Osband et al. (2013) named it PSRL and proved its guarantee. Two features matter. First, as in Thompson sampling, the agent explores in proportion to its uncertainty: a policy is followed with the posterior probability that it is optimal. Second, the sample is drawn once and kept *for the whole episode*. If the sampled MDP says that the far end of the river is wonderful, the agent swims there with determination, step after step, instead of re-deciding at each step whether to believe it. This consistency is what makes posterior sampling deep: it produces exactly the long, deliberate sequences of actions that dithering cannot.

Example 16.3 A Dirichlet posterior for the slippery Trail

On the slippery Trail of Chapter 3, taking go in Start leads to Middle or, if the hiker slips, back to Start. With a uniform prior Dirichlet(1, 1) over these two outcomes and data of 10 tries, of which 2 slipped, the posterior is Dirichlet$(1 + 2, 1 + 8)$ over (slip, advance), with mean slip probability $3/12 = 0.25$, close to the true 0.2. PSRL draws a slip probability from this posterior, perhaps 0.31 in one episode and 0.18 in the next, and plans with the drawn value. Pairs with few data have wide posteriors and produce very different plans from episode to episode; pairs with many data produce nearly the same plan every time.

♦

Figure 16.1 and Figure 16.3 show PSRL at work. On the deep sea it finds the treasure for $N = 20$ after a few hundred episodes; on RiverSwim it has the smallest regret of all the methods, about 45 after 2000 episodes. One practical detail matters a great deal: the prior. With a Dirichlet prior that puts a pseudo-count of 1 on each of the 401 possible next states of the 20×20 deep sea, a single real observation is drowned out, the sampled models are nearly uniform, and exploration slows down dramatically. The lab therefore uses $\alpha_0 = 1/S$, a prior worth one observation in total; the lab's "going further" section lets you compare. Posterior sampling inherits both the strengths and the weaknesses of the Bayesian model it uses.

INTUITION

Optimism and posterior sampling solve the same problem in two ways. An optimistic agent acts in the *best* world that is plausible given its data; a posterior-sampling agent acts in a *random* plausible world. Both visit what they are uncertain about, both stop when the uncertainty is resolved, and both commit to a plan for long enough to carry it out.

16.3.2 Randomized value functions and bootstrapped DQN

PSRL needs a posterior over MDPs and an exact planner, which are available only for small problems. The *randomized value function* idea keeps what makes PSRL work, a randomly drawn but internally consistent hypothesis followed for a whole episode, and drops the model. Instead of sampling an MDP and solving it, the agent samples a value function from an approximate posterior over value functions and acts greedily with respect to it for the episode. Randomized least-squares value iteration (RLSVI) does this for linear value functions by adding Gaussian noise to the regression targets of fitted value iteration, and has provable guarantees in the tabular case (Osband, Van Roy, et al. 2016; Osband et al. 2019).

Bootstrapped DQN (Osband, Blundell, et al. 2016) brings the idea to deep reinforcement learning. A network with a shared body and K output heads (typically $K = 10$) represents K Q-functions. Each head is trained, like DQN (Chapter 10), on its own random subset of the replay data, selected by a random mask drawn when each transition is stored, so that the heads disagree where the data are scarce. At the start of each episode one head is drawn at random, and the agent acts greedily with respect to it for the whole episode. The disagreement between the heads plays the role of posterior uncertainty, and following one head consistently produces deep exploration. Osband, Blundell, et al. (2016) reported that bootstrapped DQN explored long chains that ε-greedy DQN could not and learned faster on many Atari games.

Ensembles have a weakness: where there are no data at all, networks trained from random initializations may still agree, and then nothing drives the agent toward the unknown. *Randomized prior functions* (Osband et al. 2018) fix this by giving each head a fixed, random, untrainable prior network and training only the difference between the head's output and its prior. Where there are no data, the heads keep their random priors, and they disagree. The lab's `bootstrapped_q_learning` is a tabular version of this idea; Lab Task 3 shows that it explores the deep sea for small N but becomes unreliable as N grows, a reminder that such approximations of the posterior can be crude.

> **LAB**
>
> Lab 16.1: Deep exploration (`labs/ch16_exploration.py`, about 45 minutes). Run `python3 labs/ch16_exploration.py`. You will confirm that random exploration finds the deep-sea treasure with probability 2^{-N} and watch ε-greedy Q-learning fail; solve the deep sea with R-max, UCBVI and PSRL; compare the regret of all methods on RiverSwim and check that UCBVI's values are optimistic; tune the size of the bonus; and train random network distillation on a grid and compare prediction-error and count bonuses on a "noisy TV". The algorithms are in `rlcourse/exploration.py`.

16.4 Measuring exploration ●●

To compare exploration methods and to prove guarantees, we need to say precisely what "exploring well" means. Two measures dominate the theory. Throughout this section and the next, the setting is an *episodic MDP* with S states, A actions, horizon H, dynamics P that are the same at every step, and rewards in $[0, 1]$. The agent plays K episodes, a total of $T = KH$ steps, each

starting in a fixed state s_0; $V_0^{\pi}(s_0)$ is the expected total reward of an episode under the policy π, which may depend on the step h, and $V_0^{*}(s_0)$ is its maximum over policies.

Definition 16.1 Regret in an episodic MDP

If the agent follows policy π_k in episode k, its *regret* after K episodes is

$$\text{Regret}(K) = \sum_{k=1}^{K} \big(V_0^{*}(s_0) - V_0^{\pi_k}(s_0)\big).$$

This is the bandit regret of Section 2.1.1 with each episode as a round and each policy as an arm. It is a random variable, because the policies depend on the data; theorems bound it with high probability or in expectation. Regret that grows sublinearly in K means that the average episode becomes nearly optimal, and a bound of order $\sqrt{K}$ is the analogue of the $\sqrt{n}$ worst-case rate of bandits.

The older measure counts mistakes instead of adding up their size.

Definition 16.2 PAC sample complexity

Fix an accuracy $\varepsilon > 0$ and a confidence $\delta \in (0, 1)$. An algorithm is *PAC* (probably approximately correct) if, with probability at least $1-\delta$, the number of episodes k in which $V_0^{\pi_k}(s_0) < V_0^{*}(s_0) - \varepsilon$ is at most a bound $N(\varepsilon, \delta)$ that is polynomial in $S, A, H, 1/\varepsilon$ and $\ln(1/\delta)$. In the discounted setting the same definition counts the time steps t at which the agent's current behavior is not ε-optimal from its current state, with $1/(1-\gamma)$ in place of H; such algorithms are called *PAC-MDP*.

The discounted version is due to Kakade (2003), and Strehl et al. (2009) give a general framework for proving such bounds. E^3 (Kearns and Singh 2002), R-max (Brafman and Tennenholtz 2002) and MBIE (Strehl and Littman 2008) are PAC-MDP; model-free *delayed Q-learning* is too (Strehl et al. 2006). For R-max, Strehl et al. (2009) give a sample-complexity bound of order $S^2A/(\varepsilon^3(1-\gamma)^6)$ up to logarithmic factors.

The two measures answer different questions. A PAC bound limits the number of bad episodes but not how bad they are, and it allows an algorithm to remain ε-suboptimal forever; it suits a setting in which only the final policy matters. A regret bound limits the total loss but not the number of slightly bad episodes; it suits a setting in which every episode counts, such as a recommender serving real users. They can be converted into each other with some loss. A PAC bound $N(\varepsilon) = C/\varepsilon^2$ gives $\text{Regret}(K) \le HN(\varepsilon) + \varepsilon K$, which is of order $(CH)^{1/3}K^{2/3}$ for the best ε; conversely a regret bound $C\sqrt{K}$ implies that at most $C\sqrt{K}/\varepsilon$ episodes are ε-bad, which is not a fixed bound (Exercise 16.8). Dann et al. (2017) define *uniform-PAC* guarantees that imply both.

16.5 Regret bounds ●●

This section analyzes the optimistic algorithm UCBVI and states what is known about the best possible regret.

16.5.1 UCBVI

Upper-confidence-bound value iteration (UCBVI), from Azar et al. (2017), is the finite-horizon form of MBIE-EB: plan in the empirical model with a bonus added to every reward, and act greedily.

Algorithm 16.3 UCBVI

Input: horizon H, number of episodes K, confidence δ, bonus scale c.

1. Set all counts $N(s,a)$ and $C(s,a,s')$ to 0, and $L \leftarrow \ln(2SAHT/\delta)$ with $T = KH$.
2. For $k = 1, \dots, K$:
   1. Form the empirical model $\hat{P}(s' \mid s,a) = C(s,a,s')/N(s,a)$ and the average rewards $\hat{r}(s,a)$ of the visited pairs.
   2. Bonus: $b(s,a) \leftarrow c\,H\sqrt{L/(2N(s,a))}$.
   3. For $h = H-1, \dots, 0$: $\bar{Q}_h(s,a) \leftarrow \min\{H-h,\ \hat{r}(s,a) + \sum_{s'} \hat{P}(s' \mid s,a)\bar{V}_{h+1}(s') + b(s,a)\}$, with $\bar{Q}_h(s,a) \leftarrow H-h$ for unvisited pairs; $\bar{V}_h(s) \leftarrow \max_a \bar{Q}_h(s,a)$; $\bar{V}_H \leftarrow 0$.
   4. Follow the greedy policy $a_h = \arg\max_a \bar{Q}_h(s_h,a)$ for H steps, updating N, C and the reward averages.

Output: the policies used in the episodes.

The clipping at $H-h$ keeps the optimistic values within the range of real values. With $c = 1$, the bonus is a Hoeffding radius for an average of quantities with range H, and with probability at least $1-\delta$ it makes $\bar{Q}$ an upper bound on Q^* in every episode; the lab checks this numerically on RiverSwim. As Figure 16.3 showed, this safe choice is very conservative, and in practice smaller values of c learn much faster.

Theorem 16.3 Regret of optimistic value iteration

Consider an episodic MDP as above, with known mean rewards in $[0,1]$, and let $L = \ln(SAT/\delta)$.

(a) The variant of UCBVI whose bonus is $b(s,a) = H\sqrt{2(S\ln 2 + L)/\max\{1, N(s,a)\}}$ has, with probability at least $1-2\delta$,

$$\mathrm{Regret}(K) \le 4H\sqrt{2(S\ln 2 + L)}\left(\sqrt{2SAT} + SAH\right) + H\sqrt{2T\ln(1/\delta)},$$

which is $\tilde{O}(HS\sqrt{AT})$ for large T, where $\tilde{O}$ hides logarithmic factors.

(b) (Azar et al. 2017) UCBVI with Hoeffding bonuses of order $H/\sqrt{N(s,a)}$, up to a logarithmic factor and an absolute constant, has regret $\tilde{O}\big(H\sqrt{SAT} + H^2S^2A\big)$ with probability at least $1-\delta$; with bonuses based on Bernstein's inequality, which use the empirical variance of the next-state values, the bound becomes $\tilde{O}\big(\sqrt{HSAT} + H^2S^2A + H\sqrt{T}\big)$.

Part (b), with unknown rewards, is the result of Azar et al. (2017); we prove part (a), which shows the structure of all such proofs, and then explain the two refinements of part (b).

Proof sketch of part (a). *Step 1: a good event.* For a fixed pair (s,a) and a fixed number n of observations, the empirical distribution of the next state satisfies $\|\hat{P}(\cdot \mid s,a) - P(\cdot \mid s,a)\|_1 \le \sqrt{2(S\ln 2 + L)/n}$ with probability at least $1 - e^{-L}$, by the L_1 concentration inequality used by Jaksch et al. (2010); the reward-table argument used for Theorem 2.6 makes this valid for the random counts. A union bound over the SA pairs and the at most T values of n shows that the inequality holds for all of them at once with probability at least $1 - SATe^{-L} = 1 - \delta$. Call this the good event. On it, for every function V with values in $[0,H]$, $|(\hat{P}-P)V|(s,a) \le \|\hat{P}-P\|_1 \|V\|_\infty \le b(s,a)$.

Step 2: optimism. On the good event, $\bar{Q}_h^k \ge Q_h^*$ for every episode k and step h, by backward induction on h. If $\bar{V}_{h+1}^k \ge V_{h+1}^*$, then, since $\hat{P}$ has nonnegative entries,

$$r + \hat{P}\bar{V}_{h+1}^k + b \;\ge\; r + \hat{P}V_{h+1}^* + b \;=\; Q_h^* + (\hat{P}-P)V_{h+1}^* + b \;\ge\; Q_h^*,$$

and the clipping cannot break the inequality because $Q_h^* \le H - h$. Hence $V_0^*(s_0) \le \bar{V}_0^k(s_0)$, and the regret is at most $\sum_k (\bar{V}_0^k - V_0^{\pi_k})(s_0)$: the overestimation of the agent's own values.

Step 3: the overestimation is a sum of bonuses. Let (s_h, a_h) be the state–action pairs of episode k, with a_h the greedy action, and let $\Delta_h = (\bar{V}_h^k - V_h^{\pi_k})(s_h)$. Using the Bellman equation of π_k and Step 1 for $V = \bar{V}_{h+1}^k$,

$$\Delta_h \le b + (\hat{P}-P)\bar{V}_{h+1}^k + P\big(\bar{V}_{h+1}^k - V_{h+1}^{\pi_k}\big) \le 2b(s_h,a_h) + \Delta_{h+1} + \xi_{k,h},$$

where $\xi_{k,h} = P(\bar{V}_{h+1}^k - V_{h+1}^{\pi_k})(s_h,a_h) - \Delta_{h+1}$ is the difference between an expectation and its realization, a martingale difference bounded by H in absolute value. Unrolling over h and summing over episodes,

$$\mathrm{Regret}(K) \le 2\sum_{k=1}^{K}\sum_{h=0}^{H-1} b_k(s_h^k, a_h^k) + \sum_{k,h}\xi_{k,h}.$$

Step 4: adding up. By the Azuma–Hoeffding inequality (Theorem B.10), the martingale term is at most $H\sqrt{2T\ln(1/\delta)}$ with probability at least $1-\delta$. The bonuses add up slowly because each visit to a pair makes its next bonus smaller: grouping the visits by pair and comparing with $\int dx/\sqrt{x}$ gives the *pigeonhole bound*

$$\sum_{k,h}\frac{1}{\sqrt{\max\{1, N_k(s_h^k,a_h^k)\}}} \le 2SAH + 2\sqrt{2}\sum_{s,a}\sqrt{N_{K+1}(s,a)} \le 2SAH + 2\sqrt{2SAT},$$

where N_k counts the visits before episode k (so N_{K+1} counts all of them), the term $2SAH$ accounts for repeated visits within an episode, and the last step is the Cauchy–Schwarz inequality with $\sum_{s,a} N_{K+1}(s,a) \le T$ (Exercise 16.10). Multiplying by $2H\sqrt{2(S\ln 2 + L)}$ gives the bound. ■

The proof is a template: *optimism* turns regret into overestimation, the *Bellman equation* turns overestimation into a sum of bonuses along the visited trajectories, and the *pigeonhole bound* shows that bonuses shrinking like $1/\sqrt{N}$ add up to only $\sqrt{SAT}$. The size of the bonus determines

the final rate. Our bonus pays $\sqrt{S}$ because it controls $(\hat{P} - P)V$ for *all* value functions at once. Azar et al. (2017) instead apply Hoeffding's inequality to $(\hat{P}-P)V^*_{h+1}$ for the single fixed function V^*_{h+1}, which needs no $\sqrt{S}$, and bound the remaining term $(\hat{P} - P)(\bar{V}^k_{h+1} - V^*_{h+1})$ separately; this gives $H\sqrt{SAT}$ plus the lower-order H^2S^2A. Their Bernstein bonuses then exploit the fact that the variances of the next-state values add up to about H^2 per episode rather than H^3, which saves another $\sqrt{H}$.

16.5.2 How good can exploration be?

Lower bounds come from embedding hard bandit problems in MDPs. For episodic MDPs with time-homogeneous dynamics, every algorithm has regret of order at least $\sqrt{HSAT}$ on some MDP, a bound that Azar et al. (2017) derive from the construction of Jaksch et al. (2010); when the dynamics may differ from step to step, the lower bound grows by a factor $\sqrt{H}$ (see the discussion in Jin et al. (2018)). UCBVI with Bernstein bonuses therefore matches the lower bound up to logarithmic factors once T is large enough for the lower-order terms to be dominated, roughly $T \geq H^3S^3A$ and $SA \geq H$. Exercise 16.11 explains where $\sqrt{SAT}$ comes from.

Other algorithms have similar guarantees. UCRL2 (Jaksch et al. 2010), an optimistic algorithm for average-reward MDPs with diameter D (the longest expected time needed to travel between two states), has regret $\tilde{O}(DS\sqrt{AT})$ against a lower bound of order $\sqrt{DSAT}$. PSRL has *Bayesian* regret, the regret averaged over MDPs drawn from the prior, of order $\tilde{O}(HS\sqrt{AT})$ (Osband et al. 2013), and Russo and Van Roy (2014) show, for a broad class of bandit problems, that regret bounds proved for UCB algorithms through confidence sets carry over to the Bayesian regret of posterior sampling. Model-free methods can also explore provably: Q-learning with UCB bonuses, optimistic initialization at H, and the step size $\eta_t = (H + 1)/(H + t)$ after the t-th visit has regret $\tilde{O}(\sqrt{H^3SAT})$ with Bernstein-type bonuses (Jin et al. 2018); that analysis allows the dynamics to change from step to step, where the lower bound is of order $\sqrt{H^2SAT}$, so the bound is within a factor $\sqrt{H}$ of optimal. The unusual step size matters: the usual $1/t$ gives the earliest, most biased targets too much weight. The lab's `optimistic_q_learning` implements this algorithm; in our experiments on the deep sea and on RiverSwim it explores far more slowly than the model-based methods.

> **RESEARCH NOTE**
>
> Sharper analyses continue to appear. Regret bounds whose dominant term does not grow with the horizon when the total reward of an episode is bounded by 1 ("horizon-free" bounds) were obtained by Z. Zhang et al. (2021), and current work refines lower-order terms, variance-dependent rates and the gap-dependent, logarithmic regret that mirrors Theorem 2.6 in MDPs. A central open question is how much of this theory survives function approximation, the subject of the next section.

16.6 Exploration with function approximation ●●●

The tabular bounds grow with the number of states, which is useless when states are images. The theory of exploration with function approximation asks which structure makes efficient exploration possible.

Linear MDPs. Suppose that a known feature map $\phi(s,a) \in \mathbb{R}^d$ makes both the transitions and the rewards linear: $P(s' \mid s,a) = \phi(s,a)^\top \mu(s')$ for unknown measures μ, and $r(s,a) = \phi(s,a)^\top \theta$. Then every Q-function is linear in ϕ, and the optimistic idea carries over with least squares in place of counts. *LSVI-UCB* fits each Q_h by ridge regression of the targets $r + \max_{a'} \bar{Q}_{h+1}(s',a')$ on the features, and adds the elliptical bonus $\beta\sqrt{\phi(s,a)^\top \Lambda_h^{-1} \phi(s,a)}$ of LinUCB (Section 2.10), where Λ_h is the regularized Gram matrix of the features seen at step h. Jin, Yang, et al. (2020) prove a regret bound of order $\tilde{O}(\sqrt{d^3 H^3 T})$, independent of the numbers of states and actions.

Structural complexity measures. More general theories identify a single number that controls exploration for a class of functions. The *eluder dimension* of Russo and Van Roy (2013) measures how long a sequence of points can be in which each point's value is not determined by the values at the earlier ones; it bounds the regret of optimistic algorithms and of posterior sampling. *Bellman rank*, Bellman–eluder dimension and the decision–estimation coefficient extend the idea to reinforcement learning; Section 24.6 surveys them.

Hardness. Structure is not optional. Without assumptions, an MDP with many states can hide its rewards anywhere, and no algorithm can avoid a number of episodes that grows with the number of states; this is the deep sea in general form. More surprisingly, even when a good linear representation exists, efficient exploration can be impossible: Du et al. (2020) show that if Q^* is only approximately linear in known features, an exponential number of episodes in H (or in d) may be necessary, and Chapter 9 discusses related lower bounds for exactly linear Q^*. What makes exploration tractable is a combination of a good representation and a closedness of the function class under the Bellman operator, as in linear MDPs.

Reward-free exploration. An agent can also explore before it knows its task. In *reward-free exploration* (Jin, Krishnamurthy, et al. 2020), the agent first explores without any reward and must then be able to compute a near-optimal policy for *any* reward function given afterwards. For tabular MDPs this is possible with a number of exploration episodes polynomial in S, A, H and $1/\varepsilon$; the exploration phase must visit every state that some policy reaches with significant probability, which is also what intrinsic motivation tries to achieve heuristically.

16.7 Intrinsic motivation for deep reinforcement learning

The methods so far count visits to states. In deep reinforcement learning the agent sees images or other rich observations and essentially never sees the same one twice, so counts must be replaced by something that generalizes. The common recipe is to add an *intrinsic reward* r_t^{int} to the task's *extrinsic* reward and train any deep RL algorithm on the sum

$$r_t + \beta\, r_t^{\text{int}},$$

where r_t^{int} is large for observations that are novel or surprising and β sets its weight. The idea of rewarding an agent for curiosity is old in psychology and in artificial intelligence (Schmidhuber 2010). It became practical in deep RL around 2016, driven by one benchmark.

16.7.1 The Montezuma's Revenge story

Among the Atari games of the Arcade Learning Environment (Bellemare et al. 2013), a handful are *hard-exploration* games: the first reward requires a long, specific sequence of actions, and dithering almost never produces it. The most famous is *Montezuma's Revenge*, in which the player must climb down ladders, avoid a skull and fetch a key before the first door opens. DQN, which reached human-level play in many Atari games, scored 0 on it (Mnih et al. 2015). The game became a test bed for exploration, and the steps toward solving it trace the methods of this section.

16.7.2 Pseudo-counts and hashing

One way to generalize counts is to derive them from a density model. Let $\rho_n(x)$ be the probability that a density model trained on the first n observations assigns to observation x, and $\rho'_n(x)$ the probability it would assign after one more training step on x. For an empirical distribution, $\rho = N/n$ and $\rho' = (N+1)/(n+1)$, where N is the count of x. Bellemare et al. (2016) define a *pseudo-count* $\hat{N}(x)$ by requiring the same relations for an arbitrary model, which gives

$$\hat{N}(x) = \frac{\rho_n(x)\big(1-\rho'_n(x)\big)}{\rho'_n(x)-\rho_n(x)}$$

(Exercise 16.13). The bonus $\beta/\sqrt{\hat{N}(x)+0.01}$, with a simple pixel-level density model, let a DQN agent explore far more of Montezuma's Revenge than before; Ostrovski et al. (2017) replaced the density model by a neural one. A cheaper alternative maps observations to a short binary code with a locality-sensitive hash and counts the codes (Tang et al. 2017): similar observations share a code, so counts generalize.

16.7.3 Curiosity from prediction error

A second family rewards the agent for observations it cannot predict. The *intrinsic curiosity module* (ICM) of Pathak et al. (2017) learns a feature map ϕ of observations, trained so that the action can be predicted from $\phi(s_t)$ and $\phi(s_{t+1})$ (which makes the features focus on what the agent controls), together with a forward model $\hat{\phi}_{t+1} = f(\phi(s_t), a_t)$. The intrinsic reward is the forward model's error $\frac{1}{2}\|\hat{\phi}_{t+1} - \phi(s_{t+1})\|^2$. As the model learns a region, its errors there shrink and the agent moves on.

Prediction error has a famous failure, the *noisy-TV problem*. If some part of the environment is genuinely random, for example a television showing random images, its next observation can never be predicted, its prediction error never shrinks, and an agent that maximizes prediction error will stare at it forever. Burda, Harri Edwards, et al. (2019) demonstrated this in a large study of curiosity-driven learning: adding a noisy TV that the agent could switch to a new random channel severely slowed its exploration of a maze. Lab Task 6 shows the effect in miniature: a forward model's squared error on a noisy transition stays near the noise variance (about $\frac{1}{3}$ for noise uniform on $[-1,1]$) forever, while a count bonus $1/\sqrt{N}$ decays. The lesson is general: a good exploration signal measures the agent's *ignorance*, which can be reduced by gathering data,

not the environment's *randomness*, which cannot. Counts, posteriors and confidence bonuses measure ignorance; raw prediction error mixes both.

16.7.4 Random network distillation

Random network distillation (RND), by Burda, Harrison Edwards, et al. (2019), avoids the noisy TV with a simple trick. Fix a randomly initialized neural network f, the *target*, which maps observations to vectors and is never trained. Train a second network g, the *predictor*, to match f on the observations the agent visits, by minimizing $\|g(x) - f(x)\|^2$. The intrinsic reward of an observation is the predictor's error $\|g(x) - f(x)\|^2$. On familiar observations the predictor has been trained and the error is small; on novel ones it has not, and the error is large. Because the target is a deterministic function of the current observation, there is nothing unpredictable to chase: with enough data the error can be driven to zero everywhere, so it behaves like a generalized count.

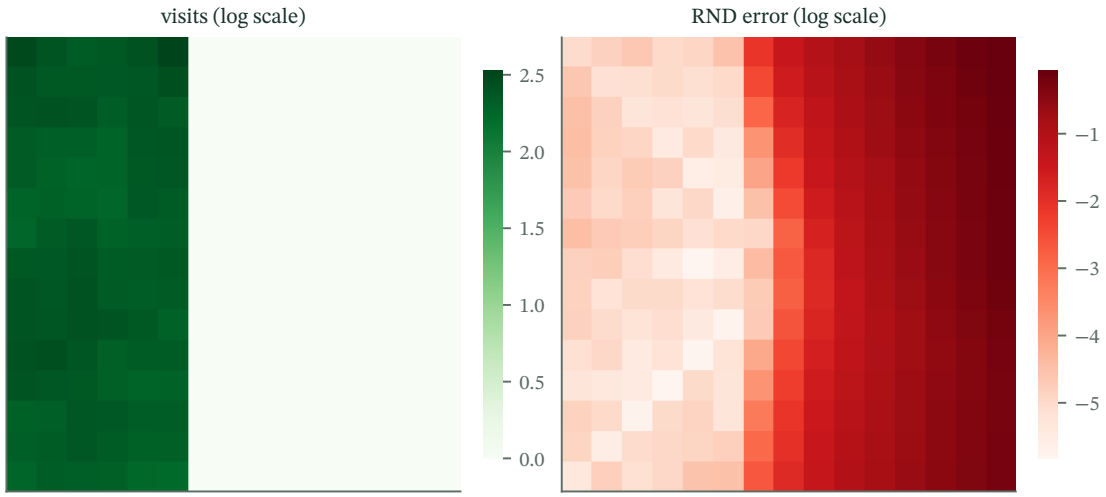


Figure 16.4. Random network distillation on a 15 × 15 grid. Left: visits of a random walk confined to the left part of the grid. Right: RND's prediction error, with the predictor trained on the visited cells only (log scale). The error is about 10^{-5} on the visited cells and about 0.23 on average elsewhere, growing with the distance from the visited region: a novelty signal that generalizes across similar states.

Figure 16.4 shows the signal on a grid, with states represented by their coordinates. With intrinsic rewards from RND, together with careful normalization of the rewards and separate value heads for intrinsic and extrinsic returns, Burda, Harrison Edwards, et al. (2019) reported the first agent to exceed average human performance on Montezuma's Revenge without demonstrations or access to the game's internal state. Later agents combined such long-term novelty with a short-term, within-episode novelty signal (Badia, Sprechmann, et al. 2020); the Agent57 agent built on this line of work and was the first to exceed the human baseline on all 57 Atari games (see Chapter 10).

16.7.5 Go-Explore

A different approach starts from a diagnosis. Ecoffet et al. (2021) argued that intrinsic-reward agents suffer from *detachment*, forgetting how to return to promising frontier regions once their bonus has been consumed nearby, and from *derailment*, the random actions that knock them off the path back to the frontier. *Go-Explore* keeps an explicit archive of the distinct situations it has reached, grouped into coarse "cells" (for example, downscaled game frames), together with the best trajectory to each. It repeatedly selects a promising cell, *returns* to it (by restoring the simulator state, or with a goal-conditioned policy), and only then explores from it with random actions, adding new cells to the archive. A final phase turns the discovered trajectories into a robust policy by imitation learning. The authors reported results far above earlier methods on Montezuma's Revenge and Pitfall and on the other hard-exploration Atari games. Go-Explore makes the point of this chapter in its own way: exploration works when the agent can *deliberately* get back to the frontier and push it outward.

PITFALL

Intrinsic rewards change the task. An agent maximizing $r + \beta r^{\text{int}}$ is not maximizing r, and with a large β it may prefer novelty over the goal; with a small β the bonus may not matter. Bonuses also help only when exploration is the bottleneck: on an easy problem with dense rewards, they mostly slow learning down. Normalize the intrinsic reward, tune β, and always compare with the agent without the bonus.

16.8 Exploration in practice

The methods of this chapter span a wide range of cost and reliability. Table 16.1 summarizes when each is the natural first choice.

Table 16.1. Choosing an exploration method.

Situation	First choice	Why
small tabular problem	UCBVI, R-max, PSRL	provable, cheap, deep
deep RL, dense rewards	ε-greedy or entropy bonus	enough, and simplest
deep RL, sparse rewards	count or RND bonus; ensembles	generalized novelty
simulator that can be reset	Go-Explore-style archive	return, then explore
random dynamics or observations	counts, posteriors	not raw prediction error

Diagnosing an exploration problem. Before adding an exploration method, check that exploration is the problem. Did the agent ever receive the reward? Log the largest return seen so far and the episodes in which any nonzero reward occurred; if the agent never sees a reward, no amount of tuning of the learning algorithm will help. Measure *coverage*: how many distinct states (or discretized cells, or hash codes) have been visited, and how that number grows. Run

a uniformly random policy and a scripted or human policy: if random play never reaches the reward but a scripted policy does, the problem is deep exploration. Finally, check whether the problem can be made easier: denser rewards through potential-based shaping, which does not change the optimal policy (Section 23.6), demonstrations (Chapter 18), or a curriculum of starting states closer to the goal.

Where to put the randomness. Dithering acts on actions, one step at a time. *Parameter-space noise* perturbs the weights of the policy network instead, once per episode, so that the agent behaves consistently within an episode, a cheap step toward the temporal consistency of posterior sampling (Plappert et al. 2018). *Noisy networks* learn how much noise to put in the weights of a Q-network and are part of Rainbow (Section 10.7). In policy-gradient and actor-critic methods, exploration comes from the stochasticity of the policy itself, and an *entropy bonus* keeps the policy from becoming deterministic too early; maximum-entropy methods such as SAC build this into the objective (Section 13.5). Entropy keeps the agent dithering; it does not make it explore deep.

Exploration in language models. Reinforcement learning of language models (Chapter 22) explores by sampling from the model's own distribution over responses, so it can only discover behaviors that the pretrained model already produces with some probability. Whether, and how, RL on verifiable rewards can find reasoning strategies beyond those of the base model is an active question (Section 22.6); it is the exploration problem of this chapter in a new guise (Exercise 16.16).

16.9 Summary

- In sequential problems the useful information may be many deliberate steps away. Dithering methods such as ε-greedy and Boltzmann exploration perturb each step independently and need a number of episodes exponential in that distance: the deep sea is found by random play with probability 2^{-N} per episode.
- Optimism in the face of uncertainty treats what the agent does not know as good. With planning, the optimism of an untried action spreads to every state that can reach it, which produces deep exploration. R-max assumes unknown pairs are maximally rewarding; count bonuses $\beta/\sqrt{N(s,a)}$ shrink gradually; UCBVI and MBIE-EB plan with the bonus.
- Theoretical bonuses guarantee optimism with high probability but are very conservative; in practice much smaller bonuses learn faster, but some optimism must survive the first visits when the dynamics are random.
- Posterior sampling (PSRL) draws one plausible MDP per episode and follows its optimal policy for the whole episode. Randomized value functions, bootstrapped DQN and ensembles with random priors approximate it when exact posteriors are not available.
- (●●) Exploration is measured by regret, the total shortfall over episodes, or by PAC sample complexity, the number of episodes that are not ε-optimal. The proof of UCBVI's regret bound follows a template: optimism, then the Bellman equation to turn overestimation into bonuses, then the pigeonhole bound. Bernstein-based UCBVI matches the $\sqrt{HSAT}$ lower bound up to logarithmic factors.
- In deep reinforcement learning, intrinsic rewards from pseudo-counts, hashing, curiosity

or random network distillation generalize counts to rich observations. Prediction error of the next observation fails on unpredictable noise (the noisy TV); good bonuses measure ignorance, not randomness.
- Before reaching for an exploration method, check that exploration is really the problem: log whether rewards are ever seen and how coverage grows.

The next chapter turns the problem around: what can an agent learn when it cannot explore at all, but must learn from data that someone else collected?

16.10 Further Reading

Sutton and Barto (2018) discuss optimistic initialization and Dyna-style exploration bonuses; Lattimore and Szepesvári (2020, ch. 38) give a rigorous introduction to regret in MDPs, and Agarwal et al. (2022) cover UCBVI and the theory of exploration with function approximation in lecture-note style. The tutorial by Russo et al. (2018) presents posterior sampling from bandits to reinforcement learning.

The foundations of provably efficient exploration are Kearns and Singh (2002) (E^3 and the known-state argument), Brafman and Tennenholtz (2002) (R-max), Strehl and Littman (2008) (MBIE and RiverSwim) and Strehl et al. (2009) (a unified PAC analysis); Kakade (2003) defined the sample complexity of exploration. For regret, start with Jaksch et al. (2010) (UCRL2) and Azar et al. (2017) (UCBVI), then Jin et al. (2018) for model-free exploration. Posterior sampling for reinforcement learning was proposed by Strens (2000) and analyzed by Osband et al. (2013); Osband et al. (2019) is a long, readable account of randomized value functions and deep exploration.

For deep reinforcement learning, the key papers are Bellemare et al. (2016) (pseudo-counts), Tang et al. (2017) (hashing), Pathak et al. (2017) (ICM), Burda, Harri Edwards, et al. (2019) (a large study of curiosity, including the noisy TV), Burda, Harrison Edwards, et al. (2019) (RND), Osband, Blundell, et al. (2016) and Osband et al. (2018) (ensembles and random priors), and Ecoffet et al. (2021) (Go-Explore). Schmidhuber (2010) reviews two decades of work on artificial curiosity.

(●●●) For exploration with function approximation, read Jin, Yang, et al. (2020) on linear MDPs, Russo and Van Roy (2013) on the eluder dimension, Du et al. (2020) on why a good representation alone is not enough, and Jin, Krishnamurthy, et al. (2020) on reward-free exploration; Dann et al. (2017) connect PAC and regret guarantees.

16.11 Exercises

Exercise 16.1 ● Dithering on the deep sea

(a) Show that a uniformly random policy finds the treasure of the deep sea of size N with probability 2^{-N} per episode, and that the expected number of episodes until it first does is 2^N.

(b) An ε-greedy agent whose greedy action is "left" in every state takes "right" with probability $\varepsilon/2$. Find the expected number of episodes until it finds the treasure, and evaluate it for $N = 10$ and $\varepsilon = 0.1$. (c) Why does the small cost of moving right make the deep sea harder for ε-greedy Q-learning than a version without that cost?

Exercise 16.2 • R-max on a long Trail

Consider the long Trail of Example 16.1 with N stations, exit rewards $1, 2, 4, \dots, 2^{N-2}$ at the first $N-1$ stations and a final reward of 2^{N-1}, no discounting and horizon $H = N$. (a) Show that R-max with $m = 1$, $r_{\max} = 2^{N-1}$ and ties broken toward exit first reaches the last station in episode N and follows the optimal policy from then on, and list the values of its policies in episodes 1 to N. (b) Redo Example 16.2 with $r_{\max} = 0.5$ on the Trail ($N = 3$). What goes wrong, and what does this say about choosing $r_{\max}$?

Exercise 16.3 • Optimistic initialization without a model

Q-learning with every $Q_h(s, a)$ initialized at its largest possible value $H - h$ and acting greedily explores the deep sea, but more slowly than R-max with $m = 1$. Explain (a) why it explores at all, (b) why optimism reaches the start of the grid only after several episodes, while R-max's planning carries it back at once, and (c) what can go wrong with optimistic initialization when the rewards or the dynamics are random.

Exercise 16.4 • The size of the theoretical bonus

On RiverSwim ($S = 6$, $A = 2$, $H = 20$) with $K = 2000$ episodes and $\delta = 0.1$, UCBVI uses $L = \ln(2SAHT/\delta)$ and the bonus $cH\sqrt{L/(2N)}$. (a) Compute L and the bonus with $c = 1$ after $N = 100$ and $N = 10\,000$ visits. (b) How many visits of a pair are needed before the bonus falls below 10% of the optimal value $V_0^*(s_0) = 3.064$? (c) Use your answers to explain the curve for $c = 1$ in Figure 16.3.

Exercise 16.5 •• How many visits make a pair known?

In a finite-horizon MDP with rewards in $[0, 1]$, suppose every known pair satisfies $\|\hat{P}(\cdot \mid s, a) - P(\cdot \mid s, a)\|_1 \le \epsilon_P$. (a) Show that for every policy π that uses only known pairs, the values in the empirical and true models differ by at most $H^2\epsilon_P/2$ at the start state. (*Hint:* write the difference as a sum over steps of $(\hat{P} - P)\hat{V}_{h+1}$ terms, as in the simulation lemma Lemma 15.1, and use $|(\hat{P} - P)V| \le \frac{1}{2}\|\hat{P} - P\|_1(\max V - \min V)$.) (b) Using the L_1 concentration bound of the proof of Theorem 16.3, find a threshold m such that, with probability at least $1 - \delta$, every pair visited m times has $\epsilon_P \le \varepsilon/H^2$. (c) What order in S, H and ε does your m have?

Exercise 16.6 • A Dirichlet posterior by hand

A state–action pair has three possible next states. With the prior Dirichlet$(1, 1, 1)$, the agent has observed the next states $2, 1, 2, 3, 2, 2$. (a) What is the posterior, and what are the posterior mean

transition probabilities? (b) With the prior Dirichlet$(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ instead? (c) Which prior reacts more strongly to the data, and why does this matter for PSRL in the 20×20 deep sea, where every pair has 401 possible next states?

Exercise 16.7 ● Why sample once per episode?

(a) Suppose PSRL drew a new MDP from its posterior at *every step* and took the first action of that MDP's optimal policy. Explain why this agent would explore the deep sea much less effectively. (b) Compare how PSRL and UCBVI choose which unknown part of the MDP to explore. When would you expect each to be better?

Exercise 16.8 ●● PAC bounds and regret

(a) Show that if at most $N(\varepsilon)$ episodes are more than ε-suboptimal, then $\text{Regret}(K) \le HN(\varepsilon) + \varepsilon K$. (b) If $N(\varepsilon) = C/\varepsilon^2$, choose ε to minimize the bound and show it is of order $(CH)^{1/3}K^{2/3}$. (c) Show that a regret bound $\text{Regret}(K) \le C\sqrt{K}$ implies that at most $C\sqrt{K}/\varepsilon$ episodes are ε-bad, and that the uniform mixture of the K policies (pick one at random) is ε-optimal in expectation once $K \ge C^2/\varepsilon^2$. (d) Explain why neither guarantee implies the other without loss.

Exercise 16.9 ●● Optimism in UCBVI

(a) Write out the induction of Step 2 in the proof of Theorem 16.3, including the base case and the role of the clipping at $H - h$. (b) Where does the proof use that the entries of $\hat{P}$ are nonnegative? (c) Explain why a bonus that only bounds $(\hat{P} - P)V^*_{h+1}$ for the fixed function V^*_{h+1} suffices for optimism, but does not suffice for Step 3 without further work.

Exercise 16.10 ●● The pigeonhole bound

Fix a pair (s, a), let n_k be its number of visits in episode k (so $n_k \le H$) and $N_k = \sum_{j<k} n_j$. (a) Show that the episodes with $N_k < H$ contribute at most $2H$ to $\sum_k n_k/\sqrt{\max\{1, N_k\}}$. (b) Show that when $N_k \ge H$, $n_k/\sqrt{N_k} \le \sqrt{2}\sum_{j=N_k+1}^{N_{k+1}} j^{-1/2}$, and conclude that $\sum_k n_k/\sqrt{\max\{1, N_k\}} \le 2H + 2\sqrt{2N_{K+1}}$. (c) Sum over pairs and use the Cauchy–Schwarz inequality to obtain the bound in the proof of Theorem 16.3.

Exercise 16.11 ●●● Where does the lower bound come from?

Build an episodic MDP that hides a bandit: from the start state, a fixed sequence of transitions leads to one of S "decision" states, uniformly at random, and in each decision state one of the A actions is slightly better than the others. (a) Argue that each decision state is a separate A-armed bandit that the agent plays about K/S times, and use the minimax lower bound for bandits of Chapter 2 (order $\sqrt{An}$ for n pulls of A arms) to obtain a regret of order $\sqrt{SAK}$ per unit of reward. (b) Explain how letting each episode's total reward scale with H, or letting the agent stay in the

decision states for many steps, turns this into the order $\sqrt{HSAT}$. (c) Read Jaksch et al. (2010) and Azar et al. (2017) and explain what differs when the dynamics may change from step to step.

Exercise 16.12 ● The noisy TV

An agent's forward model predicts the next observation o of a transition by the average of the observations seen after it so far. (a) For a deterministic transition, show that the squared prediction error is 0 after the first visit. (b) For a "noisy TV" transition whose next observation is uniform on $[-1, 1]$, show that the expected squared error after n visits is $\frac{1}{3}(1 + \frac{1}{n})$. (c) Compare with a count bonus $1/\sqrt{n}$, and explain which one an agent maximizing intrinsic reward would follow. (d) Why is RND less affected, and when could it still be fooled?

Exercise 16.13 ●● Pseudo-counts

(a) Derive the pseudo-count formula of Section 16.7 by solving $\rho = \hat{N}/\hat{n}$ and $\rho' = (\hat{N}+1)/(\hat{n}+1)$ for $\hat{N}$ and $\hat{n}$. (b) Check that for the empirical distribution ($\rho = N/n$, $\rho' = (N+1)/(n+1)$) the pseudo-count equals the true count. (c) What happens to $\hat{N}$ if the density model does not increase the probability of x after training on it ($\rho' \leq \rho$), and why must the model be "learning-positive" for pseudo-counts to make sense?

Exercise 16.14 ● Intrinsic rewards on a grid with a distractor

On a 12×12 grid world (`rlcourse.envs.GridWorld`) with the start in the top-left corner, a goal worth $+1$ in the bottom-right corner, and a "distractor" terminal cell worth $+0.1$ three cells to the right of the start, run tabular Q-learning ($\varepsilon = 0.1$, step size 0.5, $\gamma = 0.99$, episodes of at most 300 steps, 400 episodes, 5 seeds) (a) without a bonus, (b) with a count bonus $0.1/\sqrt{N(s)}$ added to the reward, and (c) with an RND bonus computed from the coordinates of the next state. Record the first episode in which the goal is reached and the number of goals in the last 100 episodes. Explain the results.

Exercise 16.15 ●● Why ensembles need random priors

(a) An ensemble of tabular Q-functions is initialized at zero, and every head is trained on the same deterministic transitions. Show that all heads stay identical, so sampling a head per episode adds no exploration. (b) With bootstrap masks the heads disagree only where data were seen. Explain why that does not drive the agent toward states it has never visited. (c) With fixed random prior tables added to each head, show that the heads disagree most where there are no data. (d) Run `bootstrapped_q_learning` on the deep sea with $N = 6$ with `prior_scale=0` and `prior_scale=1` and compare.

Exercise 16.16 ●●● Exploration in reasoning with language models

Reinforcement learning with verifiable rewards trains a language model by sampling several answers to each problem and reinforcing the correct ones (Section 22.6). (a) Explain why this

procedure can only reinforce reasoning strategies that the model already produces with nonzero probability, and relate this to dithering exploration. (b) Which ideas of this chapter could be transferred (optimism, posterior sampling, novelty bonuses, Go-Explore-style archives of partial solutions), and what would play the role of a "state" and of a "count"? (c) Propose an experiment, at a scale you could run, that would test whether a novelty bonus helps a model solve problems that it initially never solves.

CHAPTER 17

Offline Reinforcement Learning and Off-Policy Evaluation

WHAT YOU WILL LEARN

- Explain what changes when an agent must learn from a fixed data set, and define the behavior policy, coverage and concentrability.
- Explain extrapolation error, and show with numbers on the Trail why Q-learning run on a fixed data set can produce unfounded values and bad policies.
- Estimate the value of a policy from logged data with importance sampling, fitted Q evaluation and the doubly robust estimator, and compare their bias and variance.
- Describe the main families of offline methods (policy constraints, conservative values, in-sample learning) and implement TD3+BC, CQL and IQL on small problems.
- Prove that the doubly robust estimator is unbiased when either its model or its importance weights are correct (●●).
- State and prove a guarantee for pessimistic offline RL that depends only on how well the data cover a single comparison policy (●●).
- Discuss lower bounds, return-conditioned sequence models and the open problem of choosing hyperparameters without a simulator (●●●).

Every method in this book so far learns by acting. Even the off-policy methods of Chapters 5, 6, 10 and 13, which learn about one policy while following another, keep interacting with the environment and keep collecting new data. In many of the places where we would most like to use reinforcement learning, that is exactly what we cannot do. A hospital has years of records of treatments and outcomes, but it cannot try an untested dosing policy on patients to see what happens. A robotics lab has thousands of hours of logged robot data, but every new trial costs time and risks the hardware. A recommender system has billions of logged interactions, and a bad new policy deployed for a day costs real money and real users.

Offline reinforcement learning, also called *batch reinforcement learning*, learns from a fixed data set collected earlier, with no further interaction (Lange et al. 2012; Levine et al. 2020). It asks two questions. *Off-policy evaluation* (OPE) asks how good a given new policy would be. *Offline policy optimization* asks which policy we should deploy. Both must be answered from the data alone, and both run into the same obstacle: the data describe what happened under the old behavior, and a new policy may do things that the old one did rarely or never.

This obstacle changes the character of the problem. In online reinforcement learning, a wrong value estimate is a temporary nuisance: the agent acts on it, sees the consequences, and corrects

it. Offline, nothing ever corrects it. The central difficulty is therefore not exploration but its absence, and the central idea of the field is a kind of disciplined humility: trust the data where they are informative, and assume the worst, or at least nothing good, where they are silent.

Section 17.1 sets up the problem and defines coverage. Section 17.2 shows, on the Trail and on a one-state continuous problem, why ordinary off-policy algorithms fail offline. Section 17.3 treats off-policy evaluation: importance sampling, fitted Q evaluation and the doubly robust estimator, with confidence intervals (●●) and marginalized importance sampling (●●●). Section 17.4 and Section 17.5 present the main algorithm families: constraining the policy to the data (TD3+BC, advantage-weighted regression) and making the values conservative (CQL) or learning only from actions in the data (IQL). The ●● sections cover model-based offline RL (Section 17.6), the theory of pessimism (Section 17.7) and sequence-modeling approaches (Section 17.8). Section 17.9 closes with practical advice. A first reading can take Section 17.1 to 17.5 and 17.9.

17.1 Learning from a fixed data set

Imagine you are handed a log and asked to improve the policy that produced it. The log of a treatment program lists, for each patient visit, the patient's state, the treatment chosen, and what happened next. The log of a recommender lists the user's context, the item shown, and whether the user clicked. In both cases you cannot ask "what would have happened if we had done something else?" by trying it. You must answer from the log.

The data. An offline data set is a collection of transitions

$$\mathcal{D} = \{(s_i, a_i, r_i, s'_i)\}_{i=1}^{N},$$

often grouped into episodes, collected by some *behavior policy* b. (We write b, as in the notation table, for the policy that generated the data and π for the policy we want to evaluate or learn.) Sometimes b is known exactly: a randomized recommender can log the probability $b(a \mid s)$ with which it chose each item. Often it is not known: the data come from human doctors or drivers, or from a mixture of many old versions of a controller. The learner may not add to the data.

The goals. Off-policy evaluation estimates the expected return $J(\pi) = \mathbb{E}_{s_0 \sim d_0}[V^{\pi}(s_0)]$ of a given target policy π. Offline policy optimization looks for a policy with a large $J(\pi)$. Evaluation is the more basic task: if we could evaluate every policy accurately, we could optimize by comparing them, and an optimizer that cannot be checked offline is hard to trust.

What makes the offline problem different from the online problems of earlier chapters comes down to three things.

No exploration. An online agent that is unsure about an action can try it. An offline agent cannot. Whatever the behavior policy did not try remains unknown forever.

Distribution shift. The data are distributed according to the behavior policy, but the questions are about another policy. A value estimate fitted to the behavior's data is accurate where the behavior went, and possibly arbitrarily wrong elsewhere. The new policy, by design, goes somewhere else.

Coverage. Whether a question can be answered at all depends on whether the data contain the relevant situations. To make this precise, let $\mu(s, a)$ be the *data distribution*, the fraction

of the data's transitions that start with the pair (s, a) (in the limit of a large data set), and let $d^\pi(s, a) = d^\pi(s)\,\pi(a \mid s)$ be the discounted state-action visitation distribution of π Definition 3.8.

Definition 17.1 Coverage and concentrability

The data *cover* a policy π if $\mu(s, a) > 0$ for every pair (s, a) with $d^\pi(s, a) > 0$. The *concentrability coefficient* of π is

$$C^\pi = \max_{(s,a):\, d^\pi(s,a)>0} \frac{d^\pi(s, a)}{\mu(s, a)},$$

with $C^\pi = \infty$ when the data do not cover π.

In words: C^π measures how much more often π visits a state-action pair than the data do, in the worst case. If $C^\pi = 4$, then every pair that matters for π appears in the data at least a quarter as often as π would visit it. Large values of C^π mean that estimates about π rest on few samples; $C^\pi = \infty$ means that some of them rest on none.

Example 17.1 The cautious hiker's log

A cautious hiker walked the Trail ten times. At Start she went on half of the time ($b(\text{go} \mid \text{Start}) = \frac{1}{2}$), but at Middle she always exited ($b(\text{go} \mid \text{Middle}) = 0$). Her log, generated with the lab's simulator, contains seven episodes that exit at Start (return 1) and three that go on and exit at Middle (return $0 + 0.9 \cdot 2 = 1.8$), so the pairs appear with the counts

$$N(\text{Start}, \text{exit}) = 7, \quad N(\text{Start}, \text{go}) = 3, \quad N(\text{Middle}, \text{exit}) = 3, \quad N(\text{Middle}, \text{go}) = 0,$$

and Goal never appears (Figure 17.1). The log covers the policy "exit at Start" and the policy "go, then exit at Middle": we can read off their values, 1 and 1.8. It covers no policy that goes on at Middle, and in particular not the optimal one. The behavior's own value is $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1.8 = 1.4$. ♦

KEY IDEA

Offline data can only tell you about situations they contain. Every offline method is, at bottom, a decision about what to believe where the data are thin or absent.

17.2 Why off-policy learning fails offline

Q-learning is off-policy (Algorithm 6.3): its update uses the maximum over next actions, not the action the behavior took, so it can learn optimal values from any data. DQN (Algorithm 10.1) trains from a replay buffer of old transitions. It is natural to hope that running these algorithms on a fixed data set, with the data set playing the role of a replay buffer that never changes, would solve offline RL. It does not, and the reason is worth understanding in detail, because it motivates every method in the rest of the chapter.

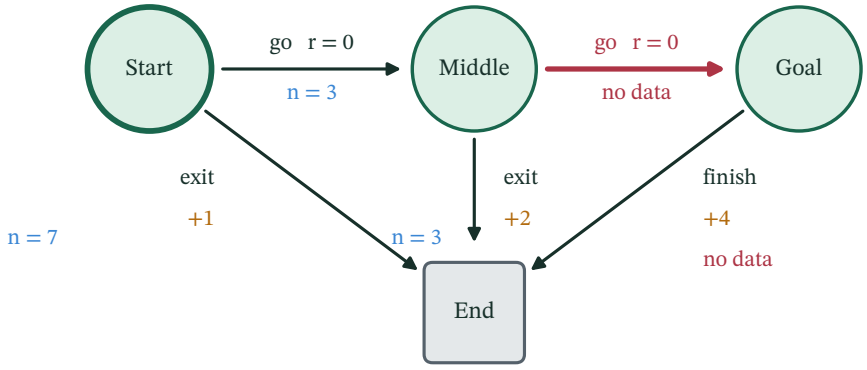


Figure 17.1. The cautious hiker's log on the Trail. The counts on the arrows say how often each state-action pair occurs in ten logged episodes. The pair (Middle, go), highlighted, never occurs, and neither does Goal: the log says nothing about what lies beyond Middle.

Example 17.2 Extrapolation error on the Trail

Run tabular Q-learning to convergence on the cautious hiker's log of Example 17.1, sweeping over the data set until nothing changes. Three pairs appear in the data and their updates settle at

$$Q(\text{Start}, \text{exit}) = 1, \qquad Q(\text{Middle}, \text{exit}) = 2,$$
$$Q(\text{Start}, \text{go}) = 0.9 \max\{Q(\text{Middle}, \text{exit}),\ Q(\text{Middle}, \text{go})\}.$$

The pair (Middle, go) never occurs, so its update never runs: $Q(\text{Middle}, \text{go})$ keeps whatever initial value q_0 it started with. Table 17.1 shows what happens for several values of q_0.

Table 17.1. Offline Q-learning on the cautious log. The unseen pair keeps its initial value q_0, which decides both the estimate and the policy.

Initial q_0	Estimate $\hat{V}(\text{Start})$	Greedy policy	True value on the Trail	True value on the broken Trail
0	1.80	go, exit	1.80	1.80
2.5	2.25	go, go	3.24	−9.00
5	4.50	go, go	3.24	−9.00
10	9.00	go, go	3.24	−9.00

With $q_0 = 0$ the agent exits at Middle and its estimate is right. With $q_0 > 2$ it goes on, and its estimate, $0.9\, q_0$, is pure invention. On the real Trail the gamble happens to pay, because Goal is worth 4. But consider a *broken Trail* in which the bridge after Middle has collapsed: going on from Middle costs −10 and ends the episode. The cautious hiker never went on from Middle, so her log would have been exactly the same in that world, and Q-learning would return exactly the same values and the same policy, now worth $0.9 \times (-10) = -9$.



Two lessons are in this small example. First, the values of actions that never appear in the data are not learned; they are *invented*, here by the initialization and in deep RL by the network's generalization. Second, no algorithm can tell the two worlds apart from this log, so any policy that goes on at Middle is a bet the data cannot back. The only policy that is good in both worlds is the one that improves where the data allow (going on at Start, worth 1.8 instead of the behavior's 1.4) and stays put where they do not.

Function approximation makes it worse. In a table, an unseen value at least stays where it was put. With function approximation, the value of an unseen action is determined by how the approximator extrapolates from the actions it was fitted on, and it changes every time the fit changes. To see the effect in its purest form, consider a problem with a single state and a continuous action $a \in [-1, 1]$. The reward is $r(a) = 1 - a^2$, so the best action is $a = 0$, the state never changes, and with $\gamma = 0.9$ the optimal value is $1/(1-\gamma) = 10$. The log contains 30 actions drawn uniformly from $[-0.3, 0.3]$, with rewards observed with Gaussian noise of standard deviation 0.1. We fit a cubic polynomial $\hat{Q}(a)$ by fitted Q iteration (Algorithm 9.6): regress on the targets $y_i = r_i + \gamma \max_{a'} \hat{Q}(a')$, where the maximum ranges over the whole action interval, and repeat.

Because the state never changes, the iteration can be solved by hand. Every target adds the same constant γM_k, where $M_k = \max_{a'} \hat{Q}_k(a')$, to the observed reward, so each fit is the least-squares cubic $\hat{f}$ of the rewards plus that constant: $\hat{Q}_{k+1}(a) = \hat{f}(a) + \gamma M_k$. Taking the maximum gives $M_{k+1} = \max_a \hat{f}(a) + \gamma M_k$, which converges to

$$M_\infty = \frac{\max_{a \in [-1,1]} \hat{f}(a)}{1-\gamma}.$$

The true value is $\max_a r(a)/(1-\gamma) = 10$. An error e in the *maximum of the fitted reward* therefore becomes an error $e/(1-\gamma) = 10\,e$ in the value: bootstrapping amplifies extrapolation error by the effective horizon. And the maximum of a cubic fitted on $[-0.3, 0.3]$ is usually far outside the data, at an end of the interval, where the cubic is free to do anything.

Figure 17.2 shows the result. For the log on the left, the fitted reward peaks at $a = 1$ with the value 3.95 (the true reward there is 0), so fitted Q iteration converges to an estimate of 39.5 for an action that is worth nothing. Over 20 logs, the greedy action ends at the boundary of the interval in 80% of them; the median estimate is 29.8 (interquartile range 24.1 to 52.3) and the median true value of the chosen action is 0. A cubic is a perfectly reasonable function class here, and it fits the data well; what fails is its use far from the data, combined with a maximization that seeks out exactly the places where it errs upward.

The mechanism. Three ingredients combine. (1) The learning target queries the value function at actions the data do not contain: the $\max_{a'}$ in Q-learning, or the actor's action in DDPG, TD3 and SAC. (2) The maximization selects the actions whose values are overestimated most, the same effect that produces maximization bias online (Section 6.7), but now without limit. (3) Bootstrapping propagates the overestimates to every state that leads to them, amplified by the horizon. Online, the agent would try the overvalued action, observe a poor outcome and correct the estimate. Offline there is no such correction. Fujimoto, Meger, et al. (2019) named this *extrapolation error* and showed that standard off-policy actor-critic algorithms can fail completely when trained on fixed data sets, even data sets generated by a good policy; Kumar et al.

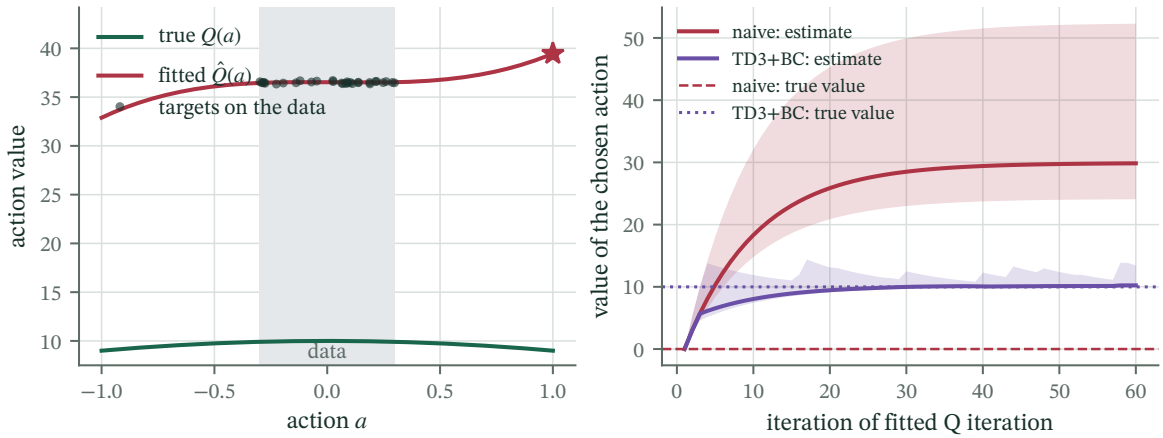


Figure 17.2. Extrapolation error in a one-state problem with a continuous action. Left: one log (dots: the regression targets), the true action values, and the cubic critic after fitted Q iteration; the critic is accurate inside the data (shaded) and its maximum (star) sits at the boundary $a = 1$, where the true reward is 0. Right: the value that each method assigns to the action it chooses, over the iterations, for 20 logs (median and interquartile band). Naive fitted Q iteration settles near 30; the actions it chooses are worth 0 (dashed). The TD3+BC actor of Section 17.4 stays near the data, and its estimate matches its true value, 10.

(2019) reported value estimates that diverge as training proceeds and called the accumulated error *bootstrapping error*.

> **OFF-POLICY IS NOT OFFLINE**
>
> An off-policy algorithm learns from data generated by another policy, but its guarantees assume that the data keep covering the actions it cares about, usually because the agent keeps exploring. On a fixed data set this assumption fails exactly where the learned policy wants to go. Running DQN, DDPG or SAC on a static log is not a safe default; it is the failure mode this chapter is about.

Every remedy in the rest of the chapter attacks one of the three ingredients. *Policy constraints* (Section 17.4) keep the policy, and hence the queried actions, close to the data. *Conservative methods* (Section 17.5) lower the values of poorly supported actions so that maximizing them is harmless. *In-sample methods* never query unseen actions at all. All of them are forms of the same principle, *pessimism in the face of uncertainty*, the mirror image of the optimism that drives exploration (Section 16.2). Section 17.7 makes the principle precise.

17.3 Off-policy evaluation

Before optimizing, we need to evaluate: given logged episodes from b and a target policy π, estimate $J(\pi)$. Evaluation matters on its own (a hospital wants a credible estimate of a new protocol's benefit before a trial), and it is the tool we would use to check any offline optimizer. There are two basic strategies. *Importance sampling* reweights the logged returns so that they look as if they came from π. The *direct method* fits a model of the environment or of Q^π and

computes $J(\pi)$ from the model. The *doubly robust* estimator combines the two.

17.3.1 Importance sampling, revisited

Chapter 5 introduced importance sampling for off-policy prediction (Section 5.4). Here is the version we need. Episode i of the log is $s_0^i, a_0^i, r_0^i, s_1^i, \ldots$, ending at time T_i, with return $G^i = \sum_{t=0}^{T_i-1} \gamma^t r_t^i$. Its per-step *importance ratios* and their running products are

$$\rho_t^i = \frac{\pi(a_t^i \mid s_t^i)}{b(a_t^i \mid s_t^i)}, \qquad \rho_{0:t}^i = \prod_{k=0}^{t} \rho_k^i.$$

The three classic estimators, for n logged episodes, are

$$\hat{J}_{\text{IS}} = \frac{1}{n} \sum_{i=1}^{n} \rho_{0:T_i-1}^i \, G^i, \qquad \hat{J}_{\text{WIS}} = \frac{\sum_i \rho_{0:T_i-1}^i \, G^i}{\sum_i \rho_{0:T_i-1}^i}, \qquad \hat{J}_{\text{PDIS}} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T_i-1} \gamma^t \rho_{0:t}^i \, r_t^i.$$

The first two are the ordinary and weighted estimators of (5.4) and (5.5), applied to the start state. The third is *per-decision* importance sampling (Section 5.5.3; Precup et al. (2000)), which weights each reward only by the ratios of the actions taken before it and including the one that earned it.

Ordinary importance sampling is unbiased because the probability of a whole episode under π divided by its probability under b is exactly $\rho_{0:T-1}$: the initial-state and transition probabilities appear in both and cancel. So $\mathbb{E}_b[\rho_{0:T-1} G] = \mathbb{E}_\pi[G] = J(\pi)$, provided the data cover π (that is, $b(a \mid s) > 0$ wherever $\pi(a \mid s) > 0$). Weighted importance sampling divides by the sum of the weights instead of by n; it is biased for finite n but consistent, and its estimate always lies between the smallest and largest observed returns.

Example 17.3 Evaluating a policy from the Trail log

A uniform hiker ($b(\text{go} \mid \cdot) = \frac{1}{2}$ at Start and at Middle) walked the Trail ten times. We want the value of the target policy π that goes on with probability 0.8 at both decisions. Its true value, from the formula of Chapter 1, is

$$J(\pi) = 0.2 \cdot 1 + 0.8 \cdot 0.2 \cdot 1.8 + 0.8 \cdot 0.8 \cdot 3.24 = 2.5616,$$

well above the behavior's 1.76. The ratios are $0.2/0.5 = 0.4$ for exit and $0.8/0.5 = 1.6$ for go (and 1 for finish, which both policies take), so each kind of episode has a fixed weight. Table 17.2 summarizes the log.

Table 17.2. The uniform hiker's log of ten Trail episodes and its importance weights for the target π ($p_0 = p_1 = 0.8$).

Episode	Count	Return G	Weight $\rho_{0:T-1}$	$\rho_{0:T-1}\, G$
exit at Start	4	1	0.4	0.4

Episode	Count	Return G	Weight $\rho_{0:T-1}$	$\rho_{0:T-1}\,G$
go, exit at Middle	3	1.8	$1.6 \times 0.4 = 0.64$	1.152
go, go, finish	3	3.24	$1.6 \times 1.6 = 2.56$	8.2944

The sum of the weighted returns is $4 \cdot 0.4 + 3 \cdot 1.152 + 3 \cdot 8.2944 = 29.9392$ and the sum of the weights is $4 \cdot 0.4 + 3 \cdot 0.64 + 3 \cdot 2.56 = 11.2$. Hence

$$\hat{J}_{\mathrm{IS}} = \frac{29.9392}{10} = 2.994, \qquad \hat{J}_{\mathrm{WIS}} = \frac{29.9392}{11.2} = 2.673.$$

Per-decision importance sampling gives exactly the ordinary estimate on the Trail: every episode earns its only nonzero reward on its last step, and that reward is weighted by the full product of ratios in both estimators. The log happens to contain more full trips (three of ten) than the behavior's average (2.5 of ten), and both estimates are too high.

♦

How reliable are such estimates? A useful diagnostic is the *effective sample size* of the weights $w_i = \rho^i_{0:T_i-1}$,

$$n_{\mathrm{eff}} = \frac{\left(\sum_i w_i\right)^2}{\sum_i w_i^2},$$

which equals n when all weights are equal and 1 when one weight dominates (Owen 2013). For the log above, $n_{\mathrm{eff}} = 11.2^2/21.5296 = 5.83$: the ten episodes carry about as much information about π as six episodes of π itself would.

The exact distributions tell the same story. One episode's ordinary estimate takes the values 0.4, 1.152 and 8.2944 with probabilities $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{4}$; its mean is 2.5616, as it should be, and its standard deviation is 3.32. With ten episodes the standard deviation is $3.32/\sqrt{10} = 1.05$. Weighted importance sampling is far less variable: in 20,000 simulated logs of ten episodes its root-mean-squared error was 0.40 (bias -0.10, standard deviation 0.39), against 1.05 for the ordinary estimator. Both errors grow with the horizon: a product of T ratios has a variance that typically grows exponentially in T (Section 5.5), which is why importance sampling alone is hopeless for long episodes.

GET THE PROBABILITIES RIGHT

Importance sampling needs the probability with which the behavior *actually* chose each logged action, including the last action of each reward, since r_t depends on a_t. Logging the action but not its probability, recording the probability under a different version of the policy, or estimating b carelessly from the data all bias the estimate, and the bias is largest exactly where b is small and the ratio is large. When the behavior is deterministic and the target is not, importance sampling cannot be used at all.

17.3.2 Fitted Q evaluation

The direct method avoids the ratios entirely. It estimates Q^π from the data and then reads off $J(\pi) = \mathbb{E}_{s_0}\big[\sum_a \pi(a \mid s_0)\,\hat{Q}(s_0, a)\big]$. The standard way to estimate Q^π from transitions is *fitted Q evaluation* (FQE): fitted Q iteration (Algorithm 9.6) with the maximum replaced by an average over the target policy's actions (Le et al. 2019).

Algorithm 17.1 Fitted Q evaluation (FQE)

Input: transitions $(s_i, a_i, r_i, s'_i, d_i)_{i=1}^N$ with $d_i = 1$ if s'_i is terminal; target policy π; a function class $\mathcal{F}$ (for example linear functions $w^\top \phi(s, a)$); number of iterations K.

1. Initialize $\hat{Q}_0 \in \mathcal{F}$, for example $\hat{Q}_0 = 0$.
2. For $k = 0, 1, \ldots, K-1$:
   1. Compute targets $y_i \leftarrow r_i + \gamma\,(1 - d_i) \sum_{a'} \pi(a' \mid s'_i)\,\hat{Q}_k(s'_i, a')$ for every i.
   2. Fit $\hat{Q}_{k+1} \leftarrow \arg\min_{f \in \mathcal{F}} \sum_{i=1}^N \big(f(s_i, a_i) - y_i\big)^2$.

Output: the estimate $\hat{J}_{\mathrm{FQE}}$, the average over the logged initial states s_0 of $\sum_a \pi(a \mid s_0)\,\hat{Q}_K(s_0, a)$.

Step 2.1 is the only place where π enters, and it enters as an expectation over the next action, so no ratio is ever computed. With a table as the function class, each regression simply averages the targets of each observed pair, and the iteration is policy evaluation in the *empirical model*, the model whose transition and reward probabilities are the frequencies in the data. With linear features its fixed point is the solution of the LSTD equations for Q^π (Algorithm 9.3).

On the uniform hiker's log, tabular FQE recovers Q^π exactly. The Trail is deterministic, so a single observation of each pair reveals its outcome: $\hat{Q}(\mathsf{Middle}, \mathsf{go}) = 0.9 \cdot 4 = 3.6$, $\hat{V}(\mathsf{Middle}) = 0.2 \cdot 2 + 0.8 \cdot 3.6 = 3.28$, $\hat{Q}(\mathsf{Start}, \mathsf{go}) = 0.9 \cdot 3.28 = 2.952$, and finally $\hat{J} = 0.2 \cdot 1 + 0.8 \cdot 2.952 = 2.5616$, the true value, from only ten episodes. When the model class is right and the data cover the target policy, the direct method can be far more accurate than any importance-sampling estimate.

Its weaknesses are the mirror image. FQE is only as good as its function class: if no member of $\mathcal{F}$ can represent Q^π, FQE converges to the wrong answer however much data it sees. Its targets query $\hat{Q}$ at the target policy's next actions, so it suffers from extrapolation error where the data do not cover π. And with function approximation it is off-policy TD learning, which can diverge (the deadly triad of Section 9.8).

17.3.3 The doubly robust estimator

Importance sampling is unbiased but noisy; the direct method is precise but biased when its model is wrong. The *doubly robust* (DR) estimator, borrowed from statistics, uses the model as a baseline and importance sampling to correct the baseline's errors.

The idea is clearest in a problem with one decision, such as a recommender choosing one item per visit (a contextual bandit, Chapter 2). With a model $\hat{Q}(s,a)$ of the expected reward and $\hat{V}(s) = \sum_a \pi(a \mid s)\,\hat{Q}(s,a)$, the DR estimate from logged triples (s_i, a_i, r_i) is (Dudík et al. 2011)

$$\hat{J}_{\mathrm{DR}} = \frac{1}{n}\sum_{i=1}^{n}\Big[\hat{V}(s_i) + \rho_i\,\big(r_i - \hat{Q}(s_i,a_i)\big)\Big].$$

In words: start from the model's prediction for π, then add an importance-weighted correction equal to the model's error on the logged action. If the model is perfect the corrections average to zero and are small, so the estimate has little variance; if the model is wrong, the corrections repair it on average.

For sequential problems, apply the same correction at every step, with the running products of ratios as weights (Jiang and Li 2016; Thomas and Brunskill 2016). For n logged episodes,

$$\hat{J}_{\mathrm{DR}} = \frac{1}{n}\sum_{i=1}^{n}\Big[\hat{V}(s_0^i) + \sum_{t=0}^{T_i-1}\gamma^t\rho_{0:t}^i\big(r_t^i + \gamma\hat{V}(s_{t+1}^i) - \hat{Q}(s_t^i,a_t^i)\big)\Big], \tag{17.1}$$

where $\hat{V}(s) = \sum_a \pi(a \mid s)\,\hat{Q}(s,a)$ and $\hat{V}(s_{T_i}^i) = 0$ when the episode ends in a terminal state. Each correction term is a temporal-difference error of the model ((6.1)), weighted by how much more likely π was than b to produce the history up to that step. With $\hat{Q} = 0$, (17.1) reduces to per-decision importance sampling; with all the weights set to zero it is the direct method.

Proposition 17.2 The doubly robust estimator is doubly robust

Suppose the logged episodes are independent draws from b. The estimator (17.1) is unbiased for $J(\pi)$ in each of the following two cases:

(a) the ratios are computed with the true behavior probabilities, π is covered by b, and $\hat{Q}$ is any fixed function (not fitted on the same episodes);

(b) $\hat{Q} = Q^\pi$, and each weight $\rho_{0:t}^i$ is any function of the history $s_0^i, a_0^i, \dots, s_t^i, a_t^i$, correct or not.

Proof. Consider a single episode and drop the index i. Let $h_t = (s_0, a_0, r_0, \dots, s_t)$ be the history up to the state at time t, and use the conventions $\rho_{0:-1} = 1$ and $\hat{V}(s_T) = 0$. Shifting the index in the $\hat{V}(s_{t+1})$ terms rewrites the bracket of (17.1) as

$$\hat{V}(s_0) + \sum_{t=0}^{T-1}\gamma^t\rho_{0:t}\big(r_t + \gamma\hat{V}(s_{t+1}) - \hat{Q}(s_t,a_t)\big) = \sum_{t=0}^{T-1}\gamma^t\Big[\rho_{0:t}\,r_t - \big(\rho_{0:t}\,\hat{Q}(s_t,a_t) - \rho_{0:t-1}\,\hat{V}(s_t)\big)\Big].$$

(The term $\hat{V}(s_0)$ on the left is the term $\rho_{0:-1}\hat{V}(s_0)$ of index 0 on the right; each term $\gamma^{t+1}\rho_{0:t}\hat{V}(s_{t+1})$ on the left reappears on the right inside the bracket of index $t+1$; and the last one, which involves s_T, vanishes because $\hat{V}(s_T) = 0$.)

Case (a). Given h_t, the only randomness in $\rho_{0:t}\hat{Q}(s_t,a_t) = \rho_{0:t-1}\,\rho_t\,\hat{Q}(s_t,a_t)$ is the action $a_t \sim b(\cdot \mid s_t)$, and $\rho_{0:t-1}$ is fixed by h_t. Hence

$$\mathbb{E}\big[\rho_{0:t}\,\hat{Q}(s_t,a_t) \mid h_t\big] = \rho_{0:t-1}\sum_a b(a \mid s_t)\,\frac{\pi(a \mid s_t)}{b(a \mid s_t)}\,\hat{Q}(s_t,a) = \rho_{0:t-1}\,\hat{V}(s_t).$$

By the tower rule each bracketed difference has expectation zero, and what remains is per-decision importance sampling, whose expectation is $J(\pi)$: $\mathbb{E}_b[\rho_{0:t} r_t] = \mathbb{E}_\pi[r_t]$ for every t, because the probability of a history up to time t under π is its probability under b multiplied by $\rho_{0:t}$.

Case (b). With $\hat{Q} = Q^\pi$ and $\hat{V} = V^\pi$, the Bellman equation (3.8) says that $\mathbb{E}[r_t + \gamma V^\pi(s_{t+1}) \mid h_t, a_t] = Q^\pi(s_t, a_t)$, where $V^\pi(s_{t+1}) = 0$ if s_{t+1} is terminal. This holds whatever policy chose a_t, because it involves only the environment. The weight $\rho_{0:t}$ is fixed given (h_t, a_t), so each correction term in the original form of (17.1) has conditional expectation zero, and the estimate's expectation is $\mathbb{E}[V^\pi(s_0)] = J(\pi)$. ■

Case (a) needs $\hat{Q}$ to be independent of the episodes it corrects. In practice one uses *cross-fitting*: fit $\hat{Q}$ on one half of the log, apply (17.1) to the other half, swap the halves and average. Neither case says anything about the estimator's variance, which is the reason to use DR at all: the closer $\hat{Q}$ is to Q^π, the smaller the corrections and the smaller the variance.

On the Trail log of Example 17.3, suppose the model is wrong in a natural way. It predicts the *behavior's* action values Q^b, as a model fitted by averaging the returns observed after each pair would: $\hat{Q}(\mathsf{Start}, \mathsf{go}) = (3 \cdot 3.24 + 3 \cdot 1.8)/6 = 2.52$ (the true $Q^\pi(\mathsf{Start}, \mathsf{go})$ is 2.952), while the values at Middle and Goal (2, 3.6, 4) happen to be right. The model alone predicts $\hat{V}(\mathsf{Start}) = 0.2 \cdot 1 + 0.8 \cdot 2.52 = 2.216$, too low. An episode that exits at Start adds the correction $0.4\,(1-1) = 0$. An episode that goes on adds $1.6\,(0 + 0.9 \cdot 3.28 - 2.52) = 0.6912$ at its first step and nothing afterwards, since the model is right from Middle on. So

$$\hat{J}_{\mathrm{DR}} = \frac{4 \cdot 2.216 + 6 \cdot 2.9072}{10} = 2.631,$$

closer to 2.5616 than either importance-sampling estimate. Its remaining error comes only from the log containing six episodes that went on at Start instead of the expected five. By exact enumeration, the one-episode DR estimate has mean 2.5616 and standard deviation 0.346, about a tenth of the ordinary estimator's 3.32, although its model describes the wrong policy.

Figure 17.3 compares the estimators on the slippery Trail, where each go fails with probability 0.2, with the same behavior and target (now $J(\pi) = 2.359$). The model must now be learned, and we also try a deliberately misspecified model: FQE with features that cannot tell Start from Middle, the state aggregation of Chapter 9. That FQE is biased and its error stalls at about 0.24. DR with the same poor model, cross-fitted as described above, is unbiased and its error keeps falling like $1/\sqrt{n}$, as Proposition 17.2 promises. Tabular FQE, with a correct function class, wins at every size from 20 episodes on. No estimator dominates in all problems: an empirical study across many domains found that the best choice depends on the horizon, on how different π is from b, and on how well the model class fits (Voloshin et al. 2021). Weighted variants of DR and estimators that blend DR with the model over different horizons (MAGIC) reduce variance further (Thomas and Brunskill 2016).

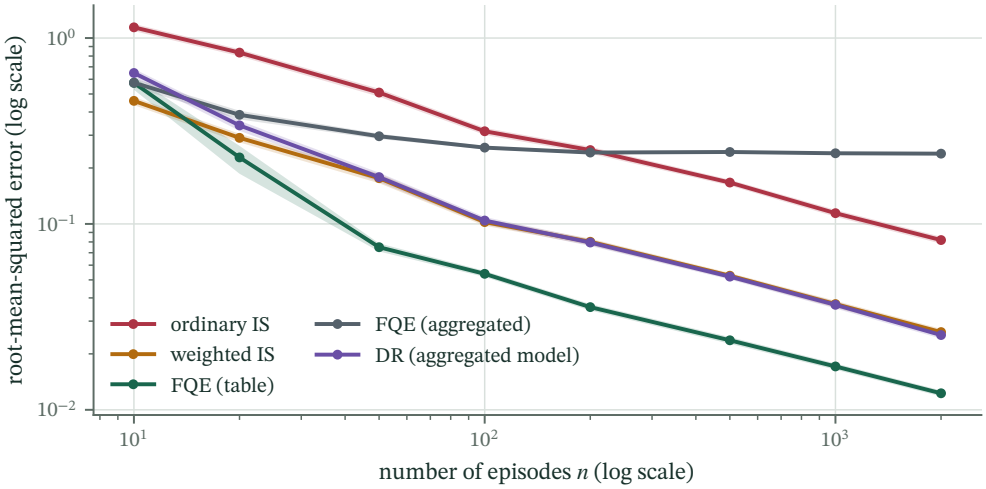


Figure 17.3. Root-mean-squared error of five off-policy estimates of $J(\pi)$ on the slippery Trail, against the number of logged episodes (log-log scales; 300 logs per size; the bands are ±1 standard error). Ordinary importance sampling is unbiased but has the largest error. Tabular FQE is best from 20 episodes on. FQE with aggregated features (Start and Middle share their values) stops improving at an error of 0.24, its bias; DR with the same misspecified model keeps improving and, from about 100 episodes on, matches weighted importance sampling.

CHECKPOINT

1. On the uniform hiker's log, what are the ordinary and weighted importance-sampling estimates of the value of the optimal policy (go, go)?
2. Why is weighted importance sampling with a single episode a poor estimate of $J(\pi)$?
3. In (17.1), what does the correction term at time t measure?

Answers.

1. Only the three full trips have nonzero weight, $2 \times 2 = 4$. Ordinary: $3 \cdot 4 \cdot 3.24/10 = 3.888$. Weighted: 3.24, exact, because every episode with nonzero weight has the same return.
2. With one episode it returns that episode's return whatever the weight, so it estimates the behavior's value (1.76 for the uniform hiker), not π's.
3. The model's temporal-difference error at the logged step, $r_t + \gamma \hat{V}(s_{t+1}) - \hat{Q}(s_t, a_t)$, reweighted by $\gamma^t \rho_{0:t}$ to the target policy's distribution.

17.3.4 Confidence intervals ●●

A point estimate is not enough to justify deploying a policy; we want an interval, or at least a credible lower bound. All the estimators above, except weighted importance sampling, are averages of independent per-episode values X_i (for ordinary importance sampling, $X_i = \rho^i_{0:T_i-1} G^i$; for DR, the bracket in (17.1)). The central limit theorem gives the usual interval $\bar{X} \pm 1.96\, \hat{\sigma}/\sqrt{n}$, and the bootstrap, which resamples whole episodes, gives intervals for any estimator, including weighted ones (Chapter 23 discusses the bootstrap in detail, Section 23.4).

These intervals assume that the sample mean is approximately normal, and importance weights

make that assumption fragile: the distribution of X_i is skewed, with rare episodes of very large weight. With few episodes, the largest weights are usually not yet in the sample, the sample variance underestimates the true variance, and the interval is too narrow exactly when the estimate is most wrong. Concentration inequalities give intervals that hold for every n. If $0 \le X_i \le x_{\max}$, Hoeffding's inequality (Theorem B.9) gives, with probability at least $1-\delta$,

$$|\bar{X} - J(\pi)| \le x_{\max}\sqrt{\frac{\ln(2/\delta)}{2n}}.$$

For ordinary importance sampling on the Trail, $x_{\max} = 8.2944$; at $n = 1000$ and $\delta = 0.05$ the half-width is 0.356, against 0.206 for the normal approximation. In longer problems $x_{\max}$ grows like the product of the largest ratios, and such bounds become useless.

Thomas et al. (2015) make high-confidence evaluation practical by a simple observation. If returns are nonnegative, clipping the weights at a threshold c can only decrease each term: $\min(\rho, c)\, G \le \rho\, G$. A lower confidence bound for the mean of the clipped terms is therefore still a lower confidence bound for $J(\pi)$, and the clipped terms are bounded by $c\, G_{\max}$, so concentration inequalities become informative. Lower bounds of this kind support *safe policy improvement*: deploy a new policy only if a high-confidence lower bound on its value exceeds the value of the current one.

17.3.5 Marginalized importance sampling ●●●

The variance of $\rho_{0:t}$ grows exponentially with t, the *curse of horizon*. Yet in a long-running process the state distribution often settles down, and what matters is not the probability of the whole history but how often π and b visit each state-action pair. This suggests weighting each transition by the *marginal* ratio

$$w(s,a) = \frac{d^\pi(s,a)}{\mu(s,a)},$$

where μ is the data distribution. Since $J(\pi) = \frac{1}{1-\gamma}\sum_{s,a} d^\pi(s,a)\, r(s,a)$ (the identity of Definition 3.8),

$$J(\pi) = \frac{1}{1-\gamma}\,\mathbb{E}_{(s,a)\sim\mu}\big[w(s,a)\, r(s,a)\big], \tag{17.2}$$

whenever the data cover π. The weight of a transition no longer depends on its past, so the variance need not grow with the horizon (Liu et al. 2018). The difficulty moves to estimating w, a ratio of two unknown distributions.

The key is that d^π satisfies a *flow equation*. For a function $\nu(s,a)$, let $(\mathcal{B}^\pi \nu)(s,a) = \gamma\, \mathbb{E}_{s'\sim P(\cdot|s,a),\, a'\sim\pi(\cdot|s')}[\nu(s',a')]$ be its discounted expected value one step later. Then for every ν,

$$\mathbb{E}_{(s,a)\sim d^\pi}\big[\nu(s,a) - (\mathcal{B}^\pi\nu)(s,a)\big] = (1-\gamma)\,\mathbb{E}_{s_0\sim d_0,\, a_0\sim\pi}\big[\nu(s_0,a_0)\big].$$

(It follows from $d^\pi = (1-\gamma)\, d_0\pi + \gamma\,(P^\pi)^\top d^\pi$, written for state-action pairs.) DualDICE (Nachum, Chow, et al. 2019) turns this into a regression. Minimize over ν

$$\tfrac{1}{2}\,\mathbb{E}_\mu\big[(\nu - \mathcal{B}^\pi\nu)(s,a)^2\big] - (1-\gamma)\,\mathbb{E}_{d_0,\pi}\big[\nu(s_0,a_0)\big].$$

Writing $x = \nu - \mathcal{B}^\pi \nu$ and using the flow equation, the objective equals $\frac{1}{2}\mathbb{E}_\mu[x^2] - \mathbb{E}_{d^\pi}[x] = \frac{1}{2}\mathbb{E}_\mu[x^2] - \mathbb{E}_\mu[w\,x]$, which is minimized pointwise at $x = w$. So the minimizer ν^* yields $w = \nu^* - \mathcal{B}^\pi \nu^*$, and the objective can be estimated from logged transitions, the initial states and π alone, without knowing b ("behavior-agnostic"). The square of an expectation inside the first term cannot be estimated without bias from single transitions (the double-sampling problem of Chapter 9); DualDICE removes it with a Fenchel-dual reformulation, which turns the problem into a min–max problem over two functions. Uehara et al. (2020) develop the minimax view in general, with weight-learning and Q-learning versions that are dual to each other.

RESEARCH NOTE

Marginalized estimators trade the curse of horizon for a hard estimation problem: density ratios of high-dimensional distributions, learned by unstable min–max optimization. How to choose the function classes, how to regularize, and how to obtain valid confidence intervals for these estimators are active questions. Lower bounds show that without assumptions such as low-variance ratios or structure in Q^π, no estimator escapes a dependence on the mismatch between π and b (Jiang and Li 2016; Wang et al. 2021b).

17.4 Keeping the policy close to the data

Extrapolation error arises when the learning target, or the policy, picks actions the data do not support. The most direct remedy is to forbid it: optimize the policy, but keep it close to the behavior, so that the values it relies on were actually fitted on data.

Behavior cloning as the baseline. The simplest offline method ignores rewards and imitates the behavior by supervised learning, maximizing $\sum_i \log \pi_\theta(a_i \mid s_i)$. Chapter 18 studies it in depth (Algorithm 18.1). It cannot improve on the behavior, but it never extrapolates, and when the data come from a good policy it is hard to beat. *Filtered* behavior cloning, which imitates only the episodes with the highest returns, is a surprisingly strong baseline on several benchmarks (L. Chen et al. 2021; Emmons et al. 2022). Every offline RL result should be compared with it.

Explicit constraints. *Batch-constrained Q-learning* (BCQ) (Fujimoto, Meger, et al. 2019) fits a generative model of the behavior's actions, samples candidate actions from it, lets a small network perturb them within a bounded range, and acts with the best candidate according to the critic; the critic's targets are computed the same way, so it is never queried far from the data. For discrete actions the idea reduces to a threshold: maximize $Q(s, a)$ only over actions whose estimated behavior probability is at least a fraction of the most likely action's (Fujimoto, Conti, et al. 2019). BEAR (Kumar et al. 2019) constrains only the *support* of the policy, by bounding a kernel distance (the maximum mean discrepancy) between the policy's and the behavior's action samples, which allows the policy to concentrate on the best supported actions rather than copying the behavior's frequencies. BRAC (Wu et al. 2019a) adds a divergence penalty $\beta\, D_{\mathrm{KL}}(\pi(\cdot \mid s) \| \hat{b}(\cdot \mid s))$ to the actor's objective or to the critic's target; its systematic comparison found that simple choices, such as a KL penalty with a well-tuned weight, perform about as well as more elaborate ones.

17.4.1 TD3+BC

The minimalist end of this family adds a behavior-cloning term to an off-policy actor-critic and changes nothing else (Fujimoto and Gu 2021). TD3 (Algorithm 13.2) trains a deterministic actor π_θ to maximize $Q(s, \pi_\theta(s))$. TD3+BC maximizes instead

$$\frac{1}{B}\sum_{i=1}^{B}\Big[\lambda\, Q\big(s_i, \pi_\theta(s_i)\big) - \big(\pi_\theta(s_i) - a_i\big)^2\Big], \qquad \lambda = \frac{\alpha}{\frac{1}{B}\sum_i |Q(s_i, a_i)|}, \tag{17.3}$$

over minibatches of logged pairs (s_i, a_i). The squared term pulls the actor towards the logged action; the first term pushes it uphill on the critic. Dividing by the average magnitude of Q makes the balance independent of the scale of the rewards, so that one value of α (the authors use 2.5) works across tasks.

Algorithm 17.2 TD3+BC

Input: a log $\mathcal{D}$ of transitions (s, a, r, s', d); critics Q_{w_1}, Q_{w_2} and actor π_θ with target copies; weight α; TD3's noise, clipping and delay parameters.

1. Normalize every state feature by its mean and standard deviation in $\mathcal{D}$.
2. Repeat for the chosen number of gradient steps:
   1. Sample a minibatch of B transitions from $\mathcal{D}$.
   2. Update both critics on TD3's clipped double-Q targets, with target-policy smoothing (Algorithm 13.2).
   3. Every few steps, update the actor by gradient ascent on (17.3), and update the target networks by Polyak averaging.

Output: the actor π_θ.

The right panel of Figure 17.2 shows the effect in the one-state problem. Using the same cubic critic, the TD3+BC actor chooses actions within 0.05 of the data mean in every log; its estimate converges to about 10.2 and the true value of its action is 9.99. The naive actor, free to roam the whole interval, believes its action is worth about 30 and is wrong by the same amount. TD3+BC is reported to be competitive with considerably more complex methods on standard benchmarks, at a fraction of their implementation and compute cost (Fujimoto and Gu 2021).

17.4.2 Advantage-weighted regression

A different way to stay close to the data is to never generate actions at all, and instead *reweight* the logged ones. Suppose we want to improve on the behavior in each state, but without moving too far from it. Measuring "too far" by the KL divergence, the improved policy should solve

$$\max_{p} \sum_{a} p(a)\, A(s, a) - \beta\, D_{\mathrm{KL}}\big(p \,\|\, b(\cdot \mid s)\big)$$

over distributions p on the actions, where A is an advantage function and $\beta > 0$ sets the strength of the constraint.

Proposition 17.3 KL-regularized improvement

For each state s the problem above has the unique solution

$$\pi^*(a \mid s) = \frac{b(a \mid s)\, \exp\big(A(s,a)/\beta\big)}{Z(s)}, \qquad Z(s) = \sum_a b(a \mid s)\, \exp\big(A(s,a)/\beta\big). \tag{17.4}$$

Proof. For any distribution p, write $\log(p/b) = \log(p/\pi^*) + \log(\pi^*/b) = \log(p/\pi^*) + A/\beta - \log Z$. Then

$$\sum_a p(a) A(s,a) - \beta \sum_a p(a) \log \frac{p(a)}{b(a \mid s)} = \beta \log Z(s) - \beta\, D_{\mathrm{KL}}\big(p \,\|\, \pi^*(\cdot \mid s)\big).$$

The first term does not depend on p, and the KL divergence is nonnegative and zero only at $p = \pi^*(\cdot \mid s)$. ∎

The solution reweights the behavior's actions by the exponentiated advantage: good actions become more likely, bad ones less, and actions the behavior never takes stay impossible. To fit a parametric policy to π^*, minimize $\mathbb{E}_s[D_{\mathrm{KL}}(\pi^*(\cdot \mid s) \| \pi_\theta(\cdot \mid s))]$. Up to a constant, this is the weighted maximum-likelihood problem

$$\max_\theta \; \mathbb{E}_{(s,a)\sim\mathcal{D}}\Big[\exp\big(A(s,a)/\beta\big) \log \pi_\theta(a \mid s)\Big],$$

where the per-state normalizer $Z(s)$ is usually dropped. This is *advantage-weighted regression* (AWR) (Peng et al. 2019): behavior cloning in which each logged action is weighted by how good it was. The idea goes back to relative-entropy policy search (Peters et al. 2010). AWAC applies it with an off-policy critic and shows that the result is a good starting point for online fine-tuning (Nair et al. 2020). As $\beta \to \infty$ the weights become equal and AWR becomes behavior cloning; as $\beta \to 0$ it puts all its weight on the best logged action in each state. The same extraction step is the last stage of implicit Q-learning in Section 17.5.3.

> **PITFALL**
>
> Constraining the policy to the behavior also limits how much it can improve. If the behavior is poor, a distribution constraint (stay close to b's action frequencies) can prevent the policy from concentrating on the rare good actions that the data do contain. Support constraints and advantage weighting are less restrictive, but no constraint helps where the data contain no good actions at all.

17.5 Conservative and in-sample methods

Policy constraints change the policy. The methods of this section change the values instead: they make the values of poorly supported actions low, so that maximizing them is harmless, or they learn values without ever evaluating actions outside the data.

17.5.1 Conservative Q-learning

Conservative Q-learning (CQL) (Kumar et al. 2020) adds a regularizer to the usual TD loss. For discrete actions its critic minimizes

$$\alpha\,\mathbb{E}_{s\sim\mathcal{D}}\Big[\log\sum_a e^{Q(s,a)} - \mathbb{E}_{a\sim\hat{b}(\cdot|s)}Q(s,a)\Big] + \frac{1}{2}\,\mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\big(Q(s,a) - r - \gamma\max_{a'}\bar{Q}(s',a')\big)^2\Big], \quad (17.5)$$

where $\hat{b}(\cdot \mid s)$ is the distribution of logged actions in s, $\bar{Q}$ is a target network and $\alpha \geq 0$ sets the strength of the regularizer. Read the first term in two parts. The log-sum-exp is a soft maximum over *all* actions: minimizing it pushes down the largest values, wherever they are. The second part, $-\mathbb{E}_{a\sim\hat{b}}Q(s,a)$, pushes up the values of the actions that actually appear in the data. The net effect lowers values that are high but poorly supported and leaves the well-supported values to the TD term.

In a table we can see exactly what the regularizer does. Setting the gradient of (17.5) with respect to $Q(s,a)$ to zero, for a pair that appears in the data with frequency $\hat{b}(a \mid s) > 0$, gives

$$Q(s,a) = y(s,a) - \alpha\Big(\frac{\mu_Q(a \mid s)}{\hat{b}(a \mid s)} - 1\Big), \qquad \mu_Q(\cdot \mid s) = \mathrm{softmax}\big(Q(s,\cdot)\big), \quad (17.6)$$

where $y(s,a)$ is the average TD target of the pair. An action that the softmax favors much more than the data do is pushed down by a large amount; an action that is common in the data is pushed up a little; an action with $\hat{b}(a \mid s) = 0$ is only pushed down. Averaging (17.6) over $a \sim \hat{b}(\cdot \mid s)$ shows that, in a state where every action appears in the data, the data-weighted average of $Q(s,\cdot)$ equals that of $y(s,\cdot)$: CQL is not biased on the data's own actions on average, only on the actions the policy would like to take instead.

A one-state example makes this concrete. Two actions appear in the data with frequencies 0.9 and 0.1, and their observed average rewards are 1.0 and 1.2; the episode ends after one step. Plain least squares would prefer the rare action. With $\alpha = 0.5$, the solution of (17.6) is $Q = (1.108, 0.231)$: the common action is raised slightly, the rare one lowered sharply, and the preference flips (Exercise 17.9 finds that it flips for every α above 0.045). On the cautious hiker's log, tabular CQL with $\alpha = 0.5$ pushes the unseen $Q(\mathrm{Middle}, \mathrm{go})$ down without limit (no data term holds it up, so it keeps falling, ever more slowly; it is at about -6.5 after the lab's default number of gradient steps) and returns the policy "go, then exit", the best policy the log supports.

Algorithm 17.3 Conservative Q-learning (CQL), discrete actions

Input: a log $\mathcal{D}$; regularizer weight α; critic Q_w with target network $Q_{\bar{w}}$; step size η.

1. Repeat for the chosen number of gradient steps:
   1. Sample a minibatch $(s_i, a_i, r_i, s'_i, d_i)_{i=1}^B$ from $\mathcal{D}$.
   2. Targets: $y_i \leftarrow r_i + \gamma\,(1 - d_i)\max_{a'} Q_{\bar{w}}(s'_i, a')$.
   3. Loss: $L(w) = \frac{1}{B}\sum_i\Big[\alpha\big(\log\sum_a e^{Q_w(s_i,a)} - Q_w(s_i,a_i)\big) + \frac{1}{2}\big(Q_w(s_i,a_i) - y_i\big)^2\Big]$.
   4. $w \leftarrow w - \eta\,\nabla_w L(w)$; update the target network.

Output: the greedy policy $\pi(s) = \arg\max_a Q_w(s, a)$.

Note that the minibatch average of $Q_w(s_i, a_i)$ in step 1.3 is the sample version of $\mathbb{E}_{a\sim\hat{b}}Q(s, a)$: the behavior distribution never has to be estimated. For continuous actions, CQL is built on soft actor-critic (Algorithm 13.3) and estimates the log-sum-exp by sampling actions from the current policy and from a uniform distribution, with importance weights.

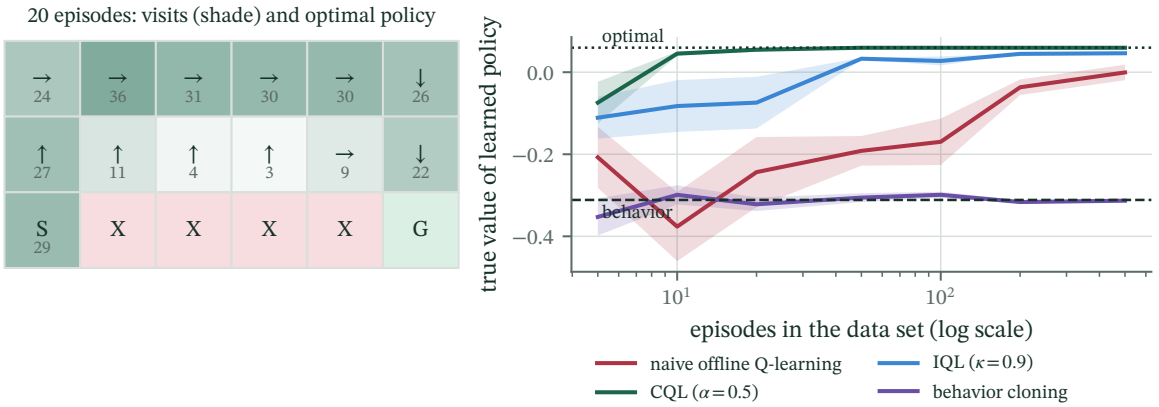


Figure 17.4. Offline control on a slippery cliff grid. Left: the grid (start S, goal G, traps X between them), shaded by how often a log of 20 episodes visits each cell, with the optimal policy, which takes the safe route along the top row. Right: the true value of the policy learned from logs of different sizes (mean of 20 logs per size; bands are ±1 standard error). The behavior follows the safe route but acts at random 30% of the time and often falls (dashed line). Behavior cloning copies its falls. Naive offline Q-learning is misled by rarely tried moves until the log holds hundreds of episodes; CQL is within 0.005 of the optimal value (dotted) from 20 episodes on, and IQL within about 0.03 from 50 on.

Figure 17.4 compares the methods on a grid where the start and the goal are separated by a row of traps and every move slips sideways with probability 0.2. The optimal policy (value 0.060) avoids the cells next to the traps by taking the top row. The behavior follows the same route but takes a random action 30% of the time, and its value is −0.311. Naive offline Q-learning, here simply value iteration in the empirical model, has a value between −0.38 and −0.17 for logs of up to 100 episodes. Its failures come from moves that the behavior tried only a few times: a move next to the traps that happened not to slip in its two or three samples looks like a safe shortcut. CQL, with $\alpha = 0.5$, is already at 0.045 with ten episodes and at the optimal value from 50 on. Behavior cloning stays at the behavior's value, −0.31, however much data it gets.

17.5.2 Why CQL is conservative ●●

The name comes from a precise property. Consider the version of CQL that evaluates a fixed policy π, with the soft maximum replaced by an expectation under π: the regularizer is $\alpha\,\mathbb{E}_{s\sim\mathcal{D}}[\mathbb{E}_{a\sim\pi}Q(s, a) - \mathbb{E}_{a\sim\hat{b}}Q(s, a)]$ and the targets use $\mathbb{E}_{a'\sim\pi}\bar{Q}(s', a')$.

Proposition 17.4 Tabular CQL underestimates the value of the evaluated policy

In a finite MDP, suppose the targets are exact Bellman backups (no sampling error), $\hat{b} = b$, and $b(a \mid s) > 0$ whenever $\pi(a \mid s) > 0$. Then the fixed point $\hat{Q}$ of the regularized evaluation satisfies, with $\hat{V}(s) = \sum_a \pi(a \mid s)\,\hat{Q}(s,a)$,

$$\hat{V} = V^\pi - \alpha\,(I - \gamma P^\pi)^{-1} D, \qquad D(s) = \sum_a \frac{\pi(a \mid s)^2}{b(a \mid s)} - 1 \;\geq\; 0.$$

So $\hat{V} \leq V^\pi$ in every state, with equality for $\alpha > 0$ only where π agrees with b along the way.

Proof. For a fixed target $y(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\,\hat{V}(s')$, the loss is a sum over states of $\alpha \sum_a (\pi(a \mid s) - b(a \mid s))\,Q(s,a) + \frac{1}{2}\sum_a b(a \mid s)(Q(s,a) - y(s,a))^2$, weighted by how often s occurs. Setting the derivative with respect to $Q(s,a)$ to zero gives $Q(s,a) = y(s,a) - \alpha\,(\pi(a \mid s)/b(a \mid s) - 1)$. Averaging over $a \sim \pi(\cdot \mid s)$ at the fixed point gives $\hat{V} = r^\pi + \gamma P^\pi \hat{V} - \alpha D$, whose solution is the displayed formula ((3.9)). By the Cauchy–Schwarz inequality, $1 = \big(\sum_a \frac{\pi(a|s)}{\sqrt{b(a|s)}}\sqrt{b(a \mid s)}\big)^2 \leq \sum_a \frac{\pi(a|s)^2}{b(a|s)} \sum_a b(a \mid s)$, so $D \geq 0$, with equality only if $\pi(\cdot \mid s) = b(\cdot \mid s)$. Finally, $(I - \gamma P^\pi)^{-1} = \sum_k \gamma^k (P^\pi)^k$ has nonnegative entries. ∎

The quantity $D(s)$ is the chi-square divergence between $\pi(\cdot \mid s)$ and $b(\cdot \mid s)$: the further the evaluated policy is from the data, the larger the underestimate. Kumar et al. (2020) prove versions of this result with sampling error, for which the lower bound holds when α is large enough relative to the statistical error, and for the full algorithm, in which the policy is improved against the conservative values. They also show that the variant without the data term, which minimizes $\mathbb{E}_{a\sim\pi} Q$ alone, gives a pointwise lower bound on Q^π rather than on V^π. Note that the full CQL objective with the soft maximum does not underestimate every value: on the cautious log it raised $Q(\mathsf{Start}, \mathsf{exit})$ to 1.18, above the true 1, because that action dominates the data.

17.5.3 Implicit Q-learning

CQL still evaluates Q at unseen actions; it just makes those values low. *Implicit Q-learning* (IQL) (Kostrikov et al. 2022) avoids them altogether. Its idea: the trouble in Q-learning's target is the $\max_{a'}$ over all actions. Replace it by a maximum over the actions that appear in the data, and estimate that maximum without enumerating the actions, by *expectile regression*.

For a random variable X and a level $\kappa \in (0,1)$, the *expectile* m_κ is the minimizer of the asymmetric squared loss

$$m_\kappa = \arg\min_m \mathbb{E}\big[L_\kappa(X - m)\big], \qquad L_\kappa(u) = \big|\kappa - \mathbb{1}\{u < 0\}\big|\,u^2. \tag{17.7}$$

(The IQL paper calls the level τ; we write κ because τ already denotes trajectories.) For $\kappa = \frac{1}{2}$ the loss is a multiple of the squared error and $m_{1/2}$ is the mean. For $\kappa > \frac{1}{2}$ the loss penalizes values of X above m more than those below, so the minimizer moves up, and as $\kappa \to 1$ it tends to the largest value X can take. Setting the derivative to zero gives the defining condition $\kappa\,\mathbb{E}[(X - m_\kappa)_+] = (1-\kappa)\,\mathbb{E}[(m_\kappa - X)_+]$.

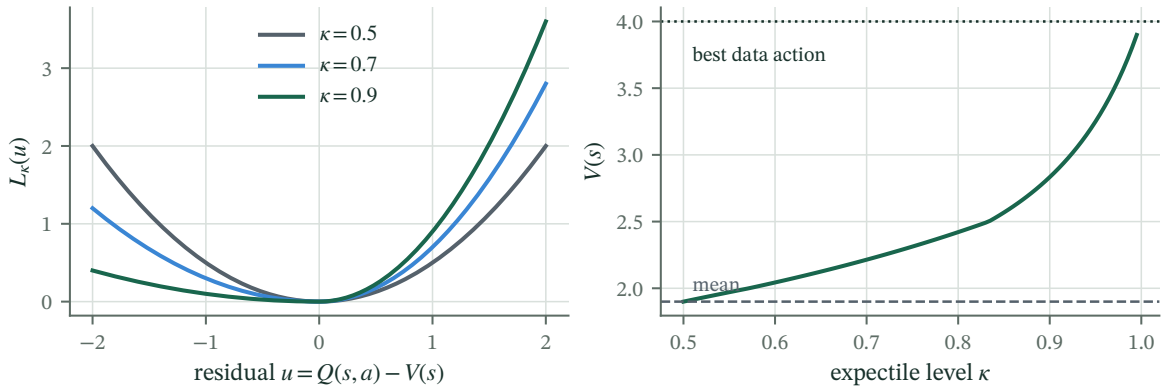


Figure 17.5. Expectile regression. Left: the asymmetric loss $L_\kappa(u)$ for three levels; for $\kappa > 0.5$ positive residuals cost more. Right: the expectile of four action values $1, 2, 2.5, 4$ taken with frequencies $0.4, 0.3, 0.2, 0.1$, as a function of κ. It starts at the mean (1.9) and rises towards the best value in the data (4) as $\kappa \to 1$, without ever using an action outside the data.

Figure 17.5 shows how the expectile of a set of action values moves from their mean towards their maximum. For the values $1, 2, 2.5, 4$ with frequencies $0.4, 0.3, 0.2, 0.1$, the expectiles at $\kappa = 0.5, 0.7, 0.9, 0.99$ are 1.9, 2.21, 2.83 and 3.81. IQL applies this to the values of the logged actions in each state.

Algorithm 17.4 Implicit Q-learning (IQL)

Input: a log $\mathcal{D}$; expectile level $\kappa \in (0.5, 1)$; temperature $\beta > 0$; networks Q_w (with target $Q_{\bar{w}}$), V_ψ and π_θ; step size η.

1. Repeat for the chosen number of gradient steps, on minibatches $(s_i, a_i, r_i, s'_i, d_i)_{i=1}^{B}$ from $\mathcal{D}$:
   1. Value step: $\psi \leftarrow \psi - \eta \nabla_\psi \frac{1}{B} \sum_i L_\kappa\big(Q_{\bar{w}}(s_i, a_i) - V_\psi(s_i)\big)$.
   2. Q step: $w \leftarrow w - \eta \nabla_w \frac{1}{B} \sum_i \big(r_i + \gamma(1 - d_i) V_\psi(s'_i) - Q_w(s_i, a_i)\big)^2$; update the target network.
2. Policy extraction: $\theta \leftarrow \arg\max_\theta \sum_i \exp\big((Q_{\bar{w}}(s_i, a_i) - V_\psi(s_i))/\beta\big) \log \pi_\theta(a_i \mid s_i)$, by gradient ascent with the weights clipped at a maximum (advantage-weighted regression, Section 17.4.2).

Output: the policy π_θ.

Every quantity in the algorithm is evaluated at logged pairs (s_i, a_i) or logged next states s'_i; no step ever asks for the value of an action outside the data. In a table, step 1.1 makes $V(s)$ the κ-expectile of the values $Q(s, a)$ of the logged actions in s, and step 1.2 makes $Q(s, a)$ the average of $r + \gamma V(s')$. As $\kappa \to 1$ the pair converges to the solution of the Bellman optimality equation restricted to the actions in the data, *in-sample Q-learning*. A level below 1 hedges against noise in the largest values, at the price of some conservatism. On the cautious hiker's log with $\kappa = 0.9$, $V(\text{Start})$ is the 0.9-expectile of 1 (frequency 0.7) and 1.8 (frequency 0.3), which is 1.635, and the

extracted policy goes on at Start and exits at Middle. In Figure 17.4, IQL with $\kappa = 0.9$ improves on the behavior from the smallest logs and is within about 0.03 of the optimum with 50 episodes or more; the reported default levels in continuous-control benchmarks are of the same order, 0.7 to 0.9 (Kostrikov et al. 2022).

LAB

Lab 17.1: Learning and evaluating from a fixed data set (`labs/ch17_offline.py`, about 45 minutes; runs in about 10 seconds). You reproduce the cautious-hiker example and the broken Trail, compute the OPE estimates of Example 17.3 together with their exact means and standard deviations, compare five estimators on the slippery Trail, and run naive offline Q-learning, CQL, IQL and behavior cloning on the cliff grid, including a sweep over CQL's weight α. The library code is in `rlcourse/offline.py`.

CHECKPOINT

1. Which term of the CQL loss would you remove to recover ordinary offline Q-learning, and what would happen on the cautious hiker's log?
2. What is the 0.5-expectile of a set of values, and what is the limit of the κ-expectile as $\kappa \to 1$?
3. Why does IQL never suffer from extrapolation error in its value targets?

Answers.

1. The regularizer (set $\alpha = 0$). The value of (Middle, go) would then stay at its initial value, and the policy would depend on that arbitrary number, as in Table 17.1.
2. The mean; and the largest value in the set (the maximum of the support).
3. Its targets $r + \gamma V(s')$ use a state value fitted only on logged state-action pairs, so Q is never evaluated at an action the data do not contain.

17.6 Model-based offline RL ●●

A learned model of the dynamics can generate new, synthetic transitions, including transitions for actions that do not appear in the data, and an agent can plan or train in the model, as in MBPO (Algorithm 15.2). Offline, this is both tempting and dangerous. Tempting, because models often generalize better than value functions: the physics of a robot is the same whatever policy collected the data. Dangerous, because an optimizer trained in a model will seek out the states where the model is wrong in its favor, which is extrapolation error again, now in the model.

The offline model-based methods counter this with pessimism in the model. MOPO (Yu et al. 2020) trains an ensemble of probabilistic dynamics models and replaces the reward by

$$\tilde{r}(s, a) = \hat{r}(s, a) - \lambda\, u(s, a),$$

where $u(s, a)$ is an uncertainty estimate (in the paper, the largest norm of a predicted covariance among the ensemble's members) and $\lambda > 0$. The policy is then optimized in the model with short rollouts started from logged states. The justification follows the simulation lemma (Lemma 15.1): the gap between a policy's value in the model and in the real environment is a discounted sum

of one-step model errors along the policy's trajectory. If the penalty $\lambda u(s, a)$ is at least as large as the effect of the model's error at every pair, then the value of any policy in the penalized model is a lower bound on its true value, and maximizing the lower bound is safe; Yu et al. (2020) make this precise. MOReL (Kidambi et al. 2020) uses the ensemble differently. Where the ensemble members disagree by more than a threshold, it declares the state-action pair unknown and redirects the transition to an absorbing "halt" state with a large negative reward; the policy learns to stay where the model is trusted.

Both methods are only as good as their uncertainty estimates. Ensemble disagreement is a heuristic measure of a model's error, not a guarantee, and the right penalty strength varies from task to task. These are the same difficulties as in the model-free methods, moved into the model.

17.7 The theory of offline RL ●●

What can be guaranteed about learning from a fixed data set, and under what conditions on the data? This section makes the principle of pessimism precise, proves its main guarantee in the tabular setting, and outlines what is known about the limits.

17.7.1 Concentrability

The classical analyses of fitted Q iteration bound the error of the learned policy by the error of the regression problems, multiplied by a concentrability coefficient (Munos and Szepesvári 2008; Chen and Jiang 2019). The coefficient that appears there is a strong one: roughly, the largest value of C^π (Definition 17.1) over *all* policies, or over the distributions reached by all sequences of policies. Such *all-policy concentrability* requires the data to cover every action in every reachable state, in proportion to how often any policy could visit it. It is plausible for data from a very exploratory behavior and implausible for data from a competent one, such as a doctor who never prescribes absurd doses. The analyses also assume that the function class is closed under the Bellman operator (*completeness*), an assumption stronger than containing Q^* (*realizability*).

The pessimistic methods of this chapter need much less. Their guarantees involve only C^π for a single comparison policy π, usually an optimal one: *single-policy concentrability*. The data need to cover what a good policy does, not everything a policy could do.

17.7.2 Pessimism

We state the guarantee for a finite discounted MDP with rewards in $[0, r_{\max}]$. From the data, form the empirical model $(\hat{r}, \hat{P})$ of Section 17.3.2 and the corresponding Bellman backup of a value function V,

$$(\hat{\mathcal{B}}V)(s, a) = \hat{r}(s, a) + \gamma \sum_{s'} \hat{P}(s' \mid s, a)\, V(s'),$$

and write $\mathcal{B}$ for the same backup with the true r and P. Choose a *penalty* $\Gamma(s, a) \geq 0$ for each pair, intended to bound the error of the empirical backup, with $\Gamma = +\infty$ for pairs that do not

appear in the data. Pessimistic value iteration computes the fixed point of

$$\hat{Q}(s,a) = \max\Big\{(\hat{\mathcal{B}}\hat{V})(s,a) - \Gamma(s,a),\ 0\Big\}, \qquad \hat{V}(s) = \max_a \hat{Q}(s,a), \tag{17.8}$$

which exists and is unique because the map is a γ-contraction in the maximum norm (taking the maximum with 0 does not increase distances), and returns the greedy policy $\hat{\pi}(s) \in \arg\max_a \hat{Q}(s,a)$.

Theorem 17.5 Pessimism competes with every well-covered policy

Let $\hat{Q}$, $\hat{V}$ and $\hat{\pi}$ be as above, and suppose the penalties are valid for the computed value function:

$$\big|(\hat{\mathcal{B}}\hat{V})(s,a) - (\mathcal{B}\hat{V})(s,a)\big| \le \Gamma(s,a) \quad \text{for all } (s,a). \tag{17.9}$$

Then (i) $\hat{V}(s) \le V^{\hat{\pi}}(s)$ for every state s: the estimate is a lower bound on the true value of the returned policy; and (ii) for every policy π,

$$J(\pi) - J(\hat{\pi}) \le \frac{2}{1-\gamma}\,\mathbb{E}_{(s,a)\sim d^{\pi}}\big[\Gamma(s,a)\big],$$

where d^{π} is the discounted state-action visitation distribution of π from d_0.

Proof. Define the *estimation error* $\iota(s,a) = (\mathcal{B}\hat{V})(s,a) - \hat{Q}(s,a)$, the true backup of $\hat{V}$ minus the estimate. We first show that $0 \le \iota \le 2\Gamma$. If the maximum in (17.8) is attained by its first argument, then $\iota = (\mathcal{B}\hat{V} - \hat{\mathcal{B}}\hat{V}) + \Gamma$, which lies in $[0, 2\Gamma]$ by (17.9). Otherwise $\hat{Q} = 0$, so $\iota = \mathcal{B}\hat{V} \ge 0$ (rewards and $\hat{V}$ are nonnegative), and $\mathcal{B}\hat{V} \le \hat{\mathcal{B}}\hat{V} + \Gamma < 2\Gamma$ because in this case $\hat{\mathcal{B}}\hat{V} - \Gamma < 0$.

(i) Since $\hat{\pi}$ is greedy, $\hat{V}(s) = \hat{Q}(s,\hat{\pi}(s)) = (\mathcal{B}\hat{V})(s,\hat{\pi}(s)) - \iota(s,\hat{\pi}(s))$. In vector form, $\hat{V} = r^{\hat{\pi}} + \gamma P^{\hat{\pi}}\hat{V} - \iota^{\hat{\pi}}$, where $\iota^{\hat{\pi}}(s) = \iota(s,\hat{\pi}(s)) \ge 0$. Subtracting the Bellman equation $V^{\hat{\pi}} = r^{\hat{\pi}} + \gamma P^{\hat{\pi}} V^{\hat{\pi}}$ gives $\hat{V} - V^{\hat{\pi}} = -(I - \gamma P^{\hat{\pi}})^{-1}\iota^{\hat{\pi}} \le 0$, because $(I - \gamma P^{\hat{\pi}})^{-1} = \sum_k \gamma^k (P^{\hat{\pi}})^k$ has nonnegative entries.

(ii) For any policy π, the greedy property gives $\hat{V}(s) \ge \sum_a \pi(a \mid s)\,\hat{Q}(s,a) = \sum_a \pi(a \mid s)\big[(\mathcal{B}\hat{V})(s,a) - \iota(s,a)\big]$, that is, $\hat{V} \ge r^{\pi} + \gamma P^{\pi}\hat{V} - \iota^{\pi}$ with $\iota^{\pi}(s) = \sum_a \pi(a \mid s)\,\iota(s,a)$. Subtracting this from $V^{\pi} = r^{\pi} + \gamma P^{\pi} V^{\pi}$ gives $V^{\pi} - \hat{V} \le \gamma P^{\pi}(V^{\pi} - \hat{V}) + \iota^{\pi}$, and iterating this inequality (the matrix P^{π} has nonnegative entries) gives $V^{\pi} - \hat{V} \le (I - \gamma P^{\pi})^{-1}\iota^{\pi}$. Averaging over $s_0 \sim d_0$ and using $d_0^{\top}(I - \gamma P^{\pi})^{-1}x = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d^{\pi}}[x(s)]$,

$$J(\pi) - \mathbb{E}_{s_0\sim d_0}[\hat{V}(s_0)] \le \frac{1}{1-\gamma}\,\mathbb{E}_{(s,a)\sim d^{\pi}}[\iota(s,a)] \le \frac{2}{1-\gamma}\,\mathbb{E}_{(s,a)\sim d^{\pi}}[\Gamma(s,a)].$$

By (i), $\mathbb{E}_{s_0}[\hat{V}(s_0)] \le J(\hat{\pi})$, and adding the two inequalities proves (ii). ■

The proof follows Y. Jin et al. (2021), who state it for finite-horizon problems with any valid penalty (their "uncertainty quantifier") and then construct such penalties for linear MDPs. Three features deserve comment.

Single-policy coverage. The right side of (ii) involves the penalties only along the comparison policy's own visitation distribution. If π visits only well-sampled pairs, the bound is small, however poorly the data cover the rest of the MDP. If π visits an unseen pair, the bound is infinite for that

π, but it still holds for every other policy. So the pessimistic policy automatically competes with the *best policy the data cover well*. On the cautious hiker's log, it competes with "go, then exit" (value 1.8) and does not attempt "go, go".

The penalties. The theorem assumes (17.9); it does not say how to achieve it. For tabular problems, Hoeffding- or Bernstein-type confidence widths of order $\frac{r_{\max}}{1-\gamma}\sqrt{L/N(s,a)}$, where $N(s,a)$ is the number of samples of the pair and L collects logarithmic factors (and, in the simplest arguments, a factor $|\mathcal{S}|$ that accounts for $\hat{V}$ depending on the data), make it hold with high probability (Rashidinejad et al. 2021). If $N(s,a) \approx N\mu(s,a)$ for N logged transitions, then $d^\pi(s,a)\,\mu(s,a)^{-1/2} \le \sqrt{C^\pi\, d^\pi(s,a)}$, and the Cauchy–Schwarz inequality gives, for a deterministic π,

$$\mathbb{E}_{d^\pi}\Big[\frac{1}{\sqrt{N\mu(s,a)}}\Big] \le \sqrt{\frac{C^\pi\,|\mathcal{S}|}{N}}.$$

The suboptimality then shrinks like $\sqrt{C^\pi|\mathcal{S}|/N}$, times a power of $1/(1-\gamma)$ and logarithmic factors. Rashidinejad et al. (2021) show in addition that pessimism adapts automatically between data from an expert, where imitation is the right strategy, and exploratory data, where it is not. For function approximation, Y. Jin et al. (2021) analyze linear MDPs and Xie et al. (2021) give a version of pessimism that constrains the value function to be consistent with the Bellman equation on the data instead of penalizing individual pairs.

Pessimism as a principle. CQL, policy constraints and MOPO can all be read as ways of obtaining a lower bound on the value and optimizing it. The theorem explains why this is the right idea, and also why it is conservative: a penalty that is too large makes $\hat{\pi}$ cling to the most frequent actions, and with large penalties near-certain early termination can look better than a long, uncertain route to a reward. The penalty is a hyperparameter with a real effect.

17.7.3 Lower bounds ●●●

Coverage of a good policy is not merely sufficient for pessimism; some coverage is necessary for any method. The Trail makes the argument concrete. The cautious log has the same distribution on the Trail and on the broken Trail, so every algorithm, however clever, outputs the same (possibly randomized) policy in both worlds. Suppose it goes on at Middle with probability q. On the Trail the best policy earns 3.24 and the output earns $1.8 + 1.44\,q$; on the broken Trail the best policy earns 1.8 and the output earns $1.8 - 10.8\,q$. The worst-case regret, $\max\{1.44(1-q),\ 10.8\,q\}$, is smallest at $q = 1.44/12.24 \approx 0.118$, where it equals 1.27. No algorithm can guarantee a regret below 1.27 in both worlds, and the pessimistic choice $q = 0$, with worst-case regret 1.44, is close to the best possible. Formal lower bounds generalize this two-world construction, and show that the dependence of pessimistic methods on C^{π^*} and N cannot be improved in general, apart from factors that depend on the horizon (Rashidinejad et al. 2021).

With function approximation the picture is starker. Wang et al. (2021b) show that, even when the value of every policy is linear in known features and the data cover all feature directions well, evaluating a policy offline can require a number of samples exponential in the horizon. Foster et al. (2022) show that realizability of the value functions together with good coverage does not suffice for sample-efficient offline learning in general; stronger assumptions on the function

class, such as Bellman completeness, or on the data are needed. An online agent could collect the data that would resolve its uncertainty; an offline agent cannot, and these results show that the difference can be very large.

RESEARCH NOTE

Open questions: Which assumptions on function classes and data make offline RL tractable, and can they be checked from the data? How should the penalty be chosen without a simulator? What is the right notion of coverage for very large state spaces, where no state is seen twice? Pessimism is minimax-optimal in the worst case, but it is not clear that the worst case is what practical data sets look like.

17.8 Offline RL as sequence modeling ●●

A different line of work drops value functions altogether and treats offline RL as supervised learning of sequences. The *Decision Transformer* (L. Chen et al. 2021) trains a transformer to predict the logged action a_t from the sequence of past states, actions and *returns-to-go* $\hat{G}_t = \sum_{t' \geq t} r_{t'}$, the reward still to be collected from time t on. At test time one conditions on a desired return, for example the best return in the data, and subtracts the rewards as they arrive. The *Trajectory Transformer* (Janner et al. 2021) models states, actions and rewards jointly and plans with beam search. These methods inherit the stability and scalability of supervised learning, and they connect offline RL to the use of large sequence models as policies (Section 24.3).

Is the transformer essential? Emmons et al. (2022) found that return-conditioned behavior cloning with a simple feed-forward network, given the right conditioning variable and enough capacity, performs comparably on several benchmarks, and called the family *RL via supervised learning* (RvS). What, then, does conditioning on a return buy, and when does it fail? The following example answers the second question.

Example 17.4 A lottery fools return conditioning

A one-step problem has two actions. Safe pays 1 for sure. Gamble pays 2 with probability 0.1 and 0 otherwise, so its expected reward is 0.2. The log comes from a uniform behavior, whose value is $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0.2 = 0.6$. Every logged episode with return 2 chose Gamble, so a return-conditioned policy asked to achieve the best logged return, 2, chooses Gamble and earns 0.2 on average, worse than the behavior. Asked for return 1 it chooses Safe and earns 1, but it has no way of knowing that 1 is the right thing to ask for. A value-based method estimates $Q(\text{Safe}) = 1$ and $Q(\text{Gamble}) = 0.2$ and chooses Safe.

♦

The failure is general: conditioning on an outcome selects the actions that were *lucky*, and in a stochastic environment luck and quality differ. Paster et al. (2022) show that Decision Transformers and RvS methods fail in stochastic environments for this reason and propose conditioning on statistics that the agent can control. Brandfonbrener et al. (2022) give conditions under

which return-conditioned supervised learning provably works, including near-deterministic dynamics and good coverage of the returns being conditioned on, and show that it can fail without them. A second limitation is *stitching*: dynamic programming can combine the first half of one logged episode with the second half of another, producing a better route than any logged episode, while conditioning on whole-episode returns has no mechanism for doing so.

17.9 Offline RL in practice

Offline to online. In many applications offline data are not the end of the story but the beginning: a robot pretrains on logs and then improves with a little real interaction. Methods that are very conservative offline can be slow to improve online, because their values for untried actions are low and the agent has no incentive to try them; less conservative ones can collapse in the first online steps. AWAC was designed for this transition (Nair et al. 2020), and Cal-QL keeps CQL's values conservative but not below the value of a reference policy such as the behavior, which is reported to speed up fine-tuning (Nakamoto et al. 2023). A simple and effective alternative is to run a sample-efficient online algorithm and mix offline transitions into every minibatch, with some care in the design of the critic (Ball et al. 2023).

Evaluation without a simulator. In a real offline application there is no environment in which to test the learned policy before deployment. The evaluation must come from off-policy estimates (Section 17.3), with honest intervals, and from a staged deployment that starts small. Two habits help. Report several estimators that rest on different assumptions (importance sampling uses the behavior's probabilities, FQE uses a model of Q), and treat disagreement between them as a warning. And report diagnostics of coverage, such as the effective sample size of the importance weights and the fraction of the learned policy's actions that are rare in the data.

Choosing hyperparameters. Every offline method has hyperparameters that matter: CQL's α, TD3+BC's α, IQL's κ and β, network sizes, numbers of gradient steps. Most published results choose them by evaluating the learned policies *in the real environment*, which a genuine offline application cannot do. Selecting among candidate policies or hyperparameters from the data alone is an open problem. Paine et al. (2020) study it on benchmark domains and report that ranking candidates with fitted Q evaluation can work reasonably in some of them; benchmarks of off-policy evaluation (Fu et al. 2021; Voloshin et al. 2021) show how much the reliability of such rankings varies across problems.

Benchmarks and their pitfalls. The D4RL suite (Fu et al. 2020) is the standard benchmark for continuous-control offline RL. It provides data sets of different quality for locomotion, navigation and manipulation tasks (from random behavior to experts, and mixtures), and reports *normalized scores*, $100 \times (\text{score} - \text{random})/(\text{expert} - \text{random})$. Results on such suites deserve the same scrutiny as any RL result (Section 23.4): many seeds, intervals rather than single numbers, and equal tuning budgets for all methods. Three pitfalls are specific to offline RL. Tuning on the evaluation environment, as above, overstates what a method would achieve offline. Data sets dominated by an expert favor behavior cloning, so averages over suites hide which method is better for which kind of data. And gains on benchmarks with deterministic or nearly deterministic dynamics may not carry over to the noisy, partially observed logs of real applications.

A short workflow. Start by understanding the log: who produced it, whether the behavior probabilities were recorded, which actions are rare. Fit behavior cloning, and filtered behavior cloning, as baselines. Then try a conservative or in-sample method (CQL, IQL or TD3+BC) and evaluate everything with more than one off-policy estimator. Prefer the simplest method whose improvement over the baselines survives the evaluation. Chapter 23 returns to the craft of RL in general (Section 23.5), and Chapter 18 to learning from demonstrations, the other main way of using logged behavior.

HISTORICAL NOTE

Learning from fixed batches of transitions is as old as approximate dynamic programming: fitted Q iteration with trees (Ernst et al. 2005) and other batch methods were developed in the 2000s (Lange et al. 2012), and the analyses of their error propagation introduced concentrability coefficients (Munos and Szepesvári 2008). The modern field took shape around 2019–2020, when extrapolation error in deep off-policy algorithms was identified (Fujimoto, Meger, et al. 2019; Kumar et al. 2019), CQL and benchmark suites appeared (Kumar et al. 2020; Fu et al. 2020), and a widely read tutorial named the open problems (Levine et al. 2020). Off-policy evaluation has a parallel history in statistics, econometrics and contextual bandits, from which the doubly robust estimator came (Dudík et al. 2011).

17.10 Summary

- Offline reinforcement learning learns from a fixed log collected by a behavior policy, with no further interaction. Its two tasks are off-policy evaluation and offline policy optimization.
- Without exploration, the data determine what can be known. Coverage and the concentrability coefficient C^π measure whether, and how well, the log describes what a policy does.
- Off-policy algorithms such as Q-learning fail offline because their targets query values of actions the data do not contain; maximization selects the overestimated ones and bootstrapping amplifies them by the horizon. On the Trail, the value of an unseen action is simply invented, and two worlds with the same log can demand opposite decisions.
- Importance sampling is unbiased but its variance grows with the horizon; the direct method (FQE) is precise when its function class is right and biased when it is not; the doubly robust estimator uses a model as a baseline and importance-weighted TD errors as corrections, and is unbiased if either the weights or the model is correct.
- Offline methods are forms of pessimism: constraining the policy to the data (BCQ, BRAC, TD3+BC, advantage-weighted regression), lowering the values of poorly supported actions (CQL), learning only from logged actions (IQL with expectile regression), or penalizing model uncertainty (MOPO, MOReL).
- (●●) Pessimistic value iteration returns a policy whose estimated value is a lower bound on its true value and whose suboptimality against any comparison policy depends only on the uncertainty along that policy's own visitation distribution: single-policy coverage suffices.
- (●●) Return-conditioned sequence models are simple and scalable, but they select lucky actions in stochastic environments and cannot stitch episodes together.

- In practice, compare with behavior cloning, evaluate with more than one estimator, and remember that choosing hyperparameters without a simulator is unsolved.

The next chapter studies the other way of learning from logged behavior: imitating it, and inferring the reward that explains it.

17.11 Further Reading

The tutorial by Levine et al. (2020) is the best entry point to offline RL as a whole: it explains distribution shift clearly, surveys the methods up to 2020 and lists open problems, many still open. For the older batch-RL literature, which developed fitted Q iteration and its relatives, see Lange et al. (2012).

On off-policy evaluation, read Dudík et al. (2011) for the doubly robust estimator in contextual bandits, then Jiang and Li (2016) and Thomas and Brunskill (2016) for its sequential versions; the latter also introduces weighted DR and MAGIC. Le et al. (2019) introduced fitted Q evaluation in its modern form, and Voloshin et al. (2021) and Fu et al. (2021) are careful empirical comparisons that are worth reading before choosing an estimator. Thomas et al. (2015) is a clear introduction to high-confidence evaluation and safe policy improvement. For marginalized importance sampling, start with Liu et al. (2018) and Nachum, Chow, et al. (2019), then Uehara et al. (2020) for the unifying minimax view.

For the algorithms, the original papers are readable and short: Fujimoto, Meger, et al. (2019) (BCQ, and the clearest demonstration of extrapolation error), Kumar et al. (2019) (BEAR), Wu et al. (2019a) (BRAC, a useful ablation study), Fujimoto and Gu (2021) (TD3+BC, a model of a simple, well-tested idea), Peng et al. (2019) and Nair et al. (2020) (advantage-weighted regression), Kumar et al. (2020) (CQL, with its lower-bound theorems), and Kostrikov et al. (2022) (IQL). For model-based methods, read Yu et al. (2020) and Kidambi et al. (2020) together; they reach similar conclusions by different routes.

(●●●) For theory, Y. Jin et al. (2021) give the cleanest statement of why pessimism works, and Rashidinejad et al. (2021) connect it to imitation learning and prove matching lower bounds in the tabular case; Xie et al. (2021) extend pessimism to general function classes. Chen and Jiang (2019) discuss which assumptions batch RL needs, and Wang et al. (2021b) and Foster et al. (2022) prove that realizability and coverage are not enough with function approximation. On sequence modeling, read L. Chen et al. (2021) and then the two critical analyses by Paster et al. (2022) and Brandfonbrener et al. (2022), which explain when return conditioning can and cannot work. For practice, D4RL (Fu et al. 2020) is the benchmark most papers use, Paine et al. (2020) is the starting point on offline hyperparameter selection, and Nakamoto et al. (2023) and Ball et al. (2023) are good entries into offline-to-online learning.

17.12 Exercises

Exercise 17.1 ● Extrapolation error by hand

Use the cautious hiker's log of Example 17.1. (a) Run offline Q-learning to convergence by hand when $Q(\text{Middle}, \text{go})$ is initialized at $q_0 = 1$, and again at $q_0 = 3$. Give the estimated $V(\text{Start})$, the greedy policy and its true value on the Trail and on the broken Trail in each case. (b) Find the range of q_0 for which the greedy policy goes on at Middle. (c) Explain why initializing all values at 0 happens to be safe on the Trail but would not be safe in a problem with negative rewards.

Exercise 17.2 • Which policies does a log cover?

For the four deterministic policies of the Trail (exit at Start; go then exit; go, go; and "exit at Start, go at Middle"), say which are covered by (a) the cautious behavior ($p_0 = 0.5$, $p_1 = 0$) and (b) the uniform behavior. (c) For a covered policy, a simple version of the concentrability coefficient is the largest ratio, over the pairs the policy uses, between the probability that the policy's episode visits the pair and the probability that a behavior episode does. Compute it for each covered policy under each behavior.

Exercise 17.3 • Evaluating the optimal policy from the log

On the uniform hiker's log of Table 17.2, compute the ordinary and weighted importance-sampling estimates of the value of the optimal policy (go, go), and the effective sample size of its weights. Explain why the weighted estimate is exact here, and why ordinary importance sampling would need many more episodes to be as accurate.

Exercise 17.4 • When per-decision weighting helps

(a) Explain why per-decision importance sampling gives the same estimate as ordinary importance sampling on every log of the Trail, slippery or not. (b) Describe a small modification of the Trail's rewards for which the two estimators differ, and say which one should have lower variance and why.

Exercise 17.5 • Doubly robust estimation with one decision

A recommender chooses between items A and B uniformly at random, $b = (0.5, 0.5)$. Four logged visits gave $(\text{A}, 1.2)$, $(\text{B}, 0.1)$, $(\text{A}, 0.8)$, $(\text{A}, 1.0)$. The target policy chooses A with probability 0.9, and a model predicts the expected rewards $\hat{Q}(\text{A}) = 0.9$, $\hat{Q}(\text{B}) = 0$. Compute the ordinary importance-sampling estimate, the model's own estimate and the doubly robust estimate of the target's value. If the true expected rewards are 1 and 0, which estimate is closest, and why are the four doubly robust terms less spread out than the four importance-sampling terms?

Exercise 17.6 •• Weighted importance sampling with one episode

(a) Show that weighted importance sampling with a single episode returns that episode's return whenever its weight is positive. (b) Deduce that, for the uniform hiker and the target with $p_0 = p_1 = 0.8$, the one-episode weighted estimate has mean 1.76 and variance 0.8368, and compare with the mean and variance of the one-episode ordinary estimate. (c) Show

that weighted importance sampling is nevertheless consistent: its estimate converges to $J(\pi)$ as the number of episodes grows.

Exercise 17.7 ●● Double robustness, step by step

(a) Verify the rearrangement used in the proof of Proposition 17.2 for an episode of length $T = 2$, writing out every term. (b) Give a counterexample showing that the doubly robust estimator can be biased when $\hat{Q}$ is fitted on the same episodes it corrects. (Hint: one decision, one logged episode, and a model that is fitted to the logged reward.) (c) Explain why cross-fitting removes this bias.

Exercise 17.8 ●● The fixed point of fitted Q evaluation

(a) Show that tabular FQE (Algorithm 17.1) converges, from any initialization, to the action-value function of π in the empirical model, for every pair that appears in the data. (b) For linear features $\phi(s, a)$, show that a fixed point w of the iteration satisfies $\sum_i \phi_i\big(\phi_i - \gamma\,\bar{\phi}'_i\big)^{\top} w = \sum_i \phi_i r_i$, where $\phi_i = \phi(s_i, a_i)$ and $\bar{\phi}'_i = (1 - d_i)\sum_{a'} \pi(a' \mid s'_i)\,\phi(s'_i, a')$; this is LSTD for Q^{π}. (c) Explain why, unlike the tabular case, the linear iteration need not converge.

Exercise 17.9 ●● CQL in a single state

One state, two actions, and episodes that end after one step. The data contain action 1 with frequency 0.9 and average reward 1.0, and action 2 with frequency 0.1 and average reward 1.2. (a) Write the tabular CQL objective (17.5) for this problem and derive the fixed-point equation (17.6). (b) Show that $0.9\,Q_1 + 0.1\,Q_2 = 1.02$ for every α. (c) Find the value of α at which the two actions tie, and solve numerically for Q when $\alpha = 0.5$.

Exercise 17.10 ● Expectiles of a coin

A random variable takes the value 1 with probability p and 0 otherwise. (a) Show that its κ-expectile is $m_\kappa = \kappa p/(\kappa p + (1 - \kappa)(1 - p))$. (b) Evaluate it for $p = 0.5$ and $\kappa = 0.5, 0.9, 0.99$, and find its limit as $\kappa \to 1$. (c) Use the defining condition to verify that the 0.9-expectile of the values 1 (frequency 0.7) and 1.8 (frequency 0.3) is 1.635, the value IQL assigns to Start on the cautious log.

Exercise 17.11 ●● Advantage weighting

(a) Derive (17.4) with a Lagrange multiplier for the constraint $\sum_a p(a) = 1$, instead of the argument in the proof of Proposition 17.3. (b) Show that projecting π^* onto a parametric family by minimizing $\mathbb{E}_s[D_{\mathrm{KL}}(\pi^*(\cdot \mid s) \| \pi_\theta(\cdot \mid s))]$ gives the weighted maximum-likelihood objective of AWR, with weights $\exp(A/\beta)/Z(s)$. (c) What happens to the extracted policy as $\beta \to 0$ and as $\beta \to \infty$, and why is clipping the weights useful in practice?

Exercise 17.12 ● Why TD3+BC normalizes

In (17.3), suppose λ were a fixed number instead of α divided by the average $|Q|$. What would happen to the balance between the two terms if all rewards were multiplied by 100? Explain how the normalization fixes this, and name one situation in which it might not be enough.

Exercise 17.13 ●● Two worlds, one log

(a) Verify the regret computation of Section 17.7.3: show that every algorithm that sees only the cautious log has worst-case regret at least 1.27 over the Trail and the broken Trail. (b) Replace the broken Trail's penalty of -10 by $-c$ with $c > 0$ and compute the best worst-case regret as a function of c. (c) How do the minimax probability of going on and the minimax regret behave as $c \to 0$ and as $c \to \infty$? Interpret.

Exercise 17.14 ●● Return conditioning and luck

(a) Reproduce the numbers of Example 17.4. (b) Change the gamble so that it pays 2 with probability p and 0 otherwise. For which p does a return-conditioned policy asked for return 2 do better than one asked for return 1? (c) Explain why the failure disappears in a deterministic environment, and describe a deterministic problem in which return conditioning still cannot find the best policy because it cannot stitch episodes.

Exercise 17.15 ● Programming: off-policy evaluation on the Trail

Extend Task 3 and Task 4 of Lab 17.1. (a) For the target policies $p_0 = p_1 \in \{0.6, 0.8, 0.95\}$, estimate the root-mean-squared error of ordinary IS, weighted IS, tabular FQE and cross-fitted DR from logs of 10, 100 and 1000 uniform-behavior episodes on the slippery Trail. (b) Plot the average effective sample size against the target, and relate it to the errors. (c) Repeat DR without cross-fitting (fitting and correcting on the same log) and compare.

Exercise 17.16 ●●● Reading the pessimism theorem

(a) On the cautious log, use penalties $\Gamma(s,a) = c/\sqrt{N(s,a)}$ for the observed pairs and $\Gamma = \infty$ otherwise, and evaluate the bound of Theorem 17.5 for the comparison policies "go, then exit" and "go, go". (b) Suppose instead that the log came from an expert who always goes all the way. Which comparison policies have a finite bound, and what does pessimism return? Contrast this with what a method requiring all-policy concentrability would need. (c) Read Rashidinejad et al. (2021) and explain in a paragraph how pessimism interpolates between imitation learning and offline RL.

Exercise 17.17 ●●● The DualDICE identity

(a) For a finite MDP, prove the flow equation of Section 17.3.5 from $d^\pi = (1-\gamma)\, d_0 \pi + \gamma\, (P^\pi)^\top d^\pi$ written for state-action pairs. (b) Show that the map $\nu \mapsto \nu - \mathcal{B}^\pi \nu$ is invertible

on functions of (s, a), and deduce that the DualDICE objective has a unique minimizer whose residual is the ratio $w = d^{\pi}/\mu$. (c) Compute w for the Trail with the uniform behavior and the target $p_0 = p_1 = 0.8$, taking μ to be the discounted visitation distribution of the behavior (including the absorbing state End), and check (17.2).

Exercise 17.18 ●●● Research: choosing hyperparameters offline

You must choose CQL's α for a medical data set with no simulator. (a) Describe three strategies for selecting α from the data alone, and the assumptions each rests on. (b) Design an experiment, using the cliff grid of Lab 17.1 (where you *can* compute true values), that measures how often each strategy picks a good α, and say what results would make you trust it on the medical data. (c) Summarize what Paine et al. (2020) and Fu et al. (2021) report about this problem and what remains open.

CHAPTER 18

Imitation Learning and Inverse Reinforcement Learning

WHAT YOU WILL LEARN

- Explain when learning from demonstrations is preferable to writing a reward, and distinguish the main settings: offline demonstrations, an interactive expert, and preferences.
- Implement behavior cloning and DAgger, and explain with numbers why the errors of behavior cloning compound over time.
- Prove the $O(\varepsilon H^2)$ bound for behavior cloning and the $O(u\varepsilon H)$ bound for DAgger (●●).
- Explain why inverse reinforcement learning is ill-posed, and how feature matching and the maximum-entropy principle resolve the ambiguity.
- Derive and implement maximum-entropy IRL, and explain adversarial imitation (GAIL) as matching occupancy measures (●●).
- Learn a reward from pairwise comparisons with the Bradley–Terry model, and explain how this leads to reinforcement learning from human feedback.
- Survey modern imitation methods for robotics and the open problems of learned rewards (●●●).

Some tasks are easy to show and hard to specify. A driving instructor can drive well but would struggle to write down a reward function whose maximization produces good driving: how many points for keeping a safe distance, how many for a smooth lane change, how many for courtesy at a four-way stop? A surgeon can demonstrate a suture; a household robot's owner can show it how to fold a shirt; a person can say which of two summaries of an article is better without being able to say why. In all these cases the most natural source of information about the task is not a reward but *behavior*: demonstrations of what to do, or judgments about which behavior is better.

This chapter studies how to learn from such information. The most direct approach, *imitation learning*, learns a policy that reproduces the demonstrated behavior. The most ambitious, *inverse reinforcement learning* (IRL), infers the reward that the demonstrator appears to be optimizing, so that the reward can be optimized, inspected, or transferred to new situations. Between them lie methods that match the demonstrator's distribution of states and actions without ever writing the reward down. And when demonstrations are unavailable, comparisons between behaviors can still teach a reward; this is the idea behind reinforcement learning from human feedback, which Chapter 22 applies to language models.

Learning from demonstrations is related to the offline RL of Chapter 17: both learn from logged

behavior without interacting with a reward. The difference is that offline RL observes rewards and tries to improve on the behavior, while imitation observes no reward and tries to match it. They share the central difficulty, distribution shift: a learner that deviates from the demonstrations finds itself in situations the demonstrations never covered.

Section 18.1 introduces the settings. Section 18.2 presents behavior cloning and proves that its errors compound quadratically in the horizon (the proof is ●●). Section 18.3 shows how an interactive expert removes the compounding (DAgger). Section 18.4 explains why inferring a reward is ill-posed and how feature matching helps; Section 18.5 derives maximum-entropy IRL; Section 18.6 treats adversarial imitation (GAIL) and its relatives (partly ●● and ●●●). Section 18.7 learns rewards from preferences with the Bradley–Terry model. Section 18.8 (●●) surveys modern imitation for robotics. A first reading can take Section 18.1, 18.2 and 18.3 without the proofs, then Section 18.4 to 18.7.

18.1 Learning from demonstrations

In 1989 Dean Pomerleau trained a small neural network, ALVINN (Autonomous Land Vehicle In a Neural Network), to map camera images of a road to steering directions (Pomerleau 1989). Nobody wrote a reward for staying on the road; the network learned from examples of the right steering for each image. ALVINN illustrated both the promise of learning from demonstrations and its central problem, which we will meet in Section 18.2: a network trained only on images of a car driven well has never seen what the road looks like when the car starts to drift, so it does not know how to recover.

The ingredients of the problem are an unknown *expert policy* π^E (a human, a planner with privileged information, or a previously trained agent), the environment, and some information about π^E. Three settings differ in what that information is.

- Offline demonstrations. A data set $\mathcal{D}_E = \{\tau_1, \dots, \tau_n\}$ of expert trajectories, each a sequence of states and actions. No rewards, and no access to the expert beyond the data. (A variant has states only, without the actions, as in videos of people.)
- An interactive expert. The learner can ask the expert what it would do in any state, including states the learner reaches by its own mistakes.
- Preferences. No demonstrations at all, but a person (or a program) who can compare two behaviors and say which is better.

There are also two broad strategies. *Direct imitation* learns a policy from the information, as a supervised learning problem or close to one. *Reward inference* first learns a reward that explains the information and then optimizes it with reinforcement learning. The second is more work, but a reward is a compact and transferable description of a task: a reward for "reach the goal without touching the water" still applies when the map changes, while a policy learned on one map does not.

Throughout the chapter, the analysis uses a finite horizon H, the length of an episode, and writes $J(\pi)$ for the expected total reward over H steps, with rewards in $[0, 1]$; the reward exists in the analysis even when the learner cannot see it. A running example makes things concrete.

Example 18.1 The windy lane

A car drives along a straight road. Its lateral position is $y \in \{-5, \dots, 5\}$, and at each step it steers left ($y - 1$), goes straight, or steers right ($y + 1$). After each action a gust pushes the car one more cell to the right with probability 0.3. Landing at position y earns $1 - |y|/6$, so the center is best, and leaving the road ($|y| > 5$) ends the episode. The expert, computed by value iteration, steers towards the center: left when $y > 0$, straight at $y = 0$, right when $y < 0$. Starting from the center, it alternates between $y = 0$ (71% of the time) and $y = 1$ (29%), and never visits any other position; over $H = 50$ steps it earns 47.5. Its demonstrations contain only two kinds of states.

♦

18.2 Behavior cloning

The simplest approach treats imitation as supervised learning: the states are inputs, the expert's actions are labels, and the policy is a classifier (or, for continuous actions, a regression model) fitted by maximum likelihood.

Algorithm 18.1 Behavior cloning

Input: expert demonstrations $\mathcal{D}_E$, flattened into state-action pairs $(s_i, a_i)_{i=1}^N$; a policy class π_θ.

1. Fit $\theta \leftarrow \arg\max_\theta \sum_{i=1}^N \log \pi_\theta(a_i \mid s_i)$, possibly with regularization, by any supervised-learning method.

Output: the policy π_θ.

For discrete actions, the objective is the usual cross-entropy loss of classification; for a Gaussian policy with fixed variance it is the squared error between the predicted and the demonstrated action. Behavior cloning (BC) needs no reward, no interaction and no model, and with enough data and a good policy class it can work remarkably well. It is also the right baseline for every other method in this chapter and in Chapter 17.

On the windy lane, a tabular softmax policy fitted to 10 expert episodes of 50 steps (with a little weight decay, so that its probabilities are not exactly 0 or 1, as with any learned policy) agrees with the expert's action on the expert's states 96.8% of the time: its error rate is $\epsilon = 0.032$. A supervised learner that makes a mistake with probability 0.032 on each of 50 decisions makes $\epsilon H = 1.6$ mistakes per episode on average, and if each mistake cost at most one step's reward we would expect to lose at most about 1.6. The cloned policy actually earns 40.5 against the expert's 47.5, a loss of 7.0, and it leaves the road in 27% of its episodes (Figure 18.1).

What went wrong is the core issue of imitation learning. A supervised learner is evaluated on the same distribution it was trained on. A cloned policy is trained on the expert's states but evaluated

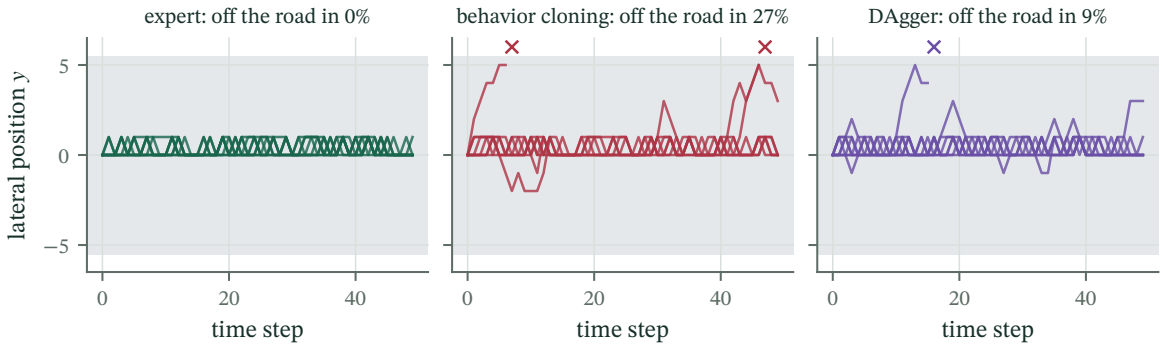


Figure 18.1. Compounding errors on the windy lane. Each panel shows eight rollouts of 50 steps; the road is shaded and a cross marks a car that left it. The expert (left) stays at $y = 0$ or 1. Behavior cloning (middle), trained on 10 expert episodes, occasionally makes a mistake, is blown to positions it has never seen demonstrated, behaves randomly there and drifts off the road in 27% of episodes. DAgger (right), trained with the same number of expert-labeled episodes, has learned to steer back.

on *its own* states. A mistake, helped by a gust, can put the car at $y = 2$ (or at $y = -1$), a position the expert never visited, so the demonstrations say nothing about what to do there. The cloned policy acts at random, the wind pushes it further right, and one mistake turns into many. The errors *compound*.

18.2.1 Errors compound quadratically

The following theorem makes the compounding precise (Ross and Bagnell 2010). Let d_t^E be the distribution of the expert's state at time t, and let

$$\epsilon = \frac{1}{H} \sum_{t=0}^{H-1} \mathbb{E}_{s \sim d_t^E} \left[\pi(a \neq \pi^E(s) \mid s) \right]$$

be the probability that the learner's action differs from the deterministic expert's, averaged over the expert's own states. This is exactly the quantity that supervised learning controls.

Theorem 18.1 Behavior cloning loses at most ϵH^2

In a problem with horizon H and rewards in $[0, 1]$, let π^E be a deterministic expert and π any policy with error ϵ on the expert's state distribution, as defined above. Then

$$J(\pi^E) - J(\pi) \leq \epsilon H^2.$$

For every ϵ and H with $\epsilon H \leq 1$ there is a problem and a policy with error ϵ for which $J(\pi^E) - J(\pi) \geq \frac{1}{2} \epsilon H(H-1)(1-\epsilon H)$, so the quadratic dependence on H cannot be improved in general.

Proof. *Upper bound.* Build the expert's and the learner's trajectories on one probability space, so that they coincide for as long as possible. Run the expert. At each time t, as long as the two trajectories have not yet separated, draw the learner's action from $\pi(\cdot \mid s_t)$; if it equals the expert's action $\pi^E(s_t)$, both take it and both move to the same next state; otherwise the trajectories

separate at time t and continue independently from then on, each with its own policy. Each trajectory, taken alone, has the right distribution. Let τ be the time of separation ($\tau = H$ if it never happens).

Before τ the two trajectories are identical and earn identical rewards. From time τ on, the expert's reward exceeds the learner's by at most 1 per step, because rewards lie in $[0, 1]$. Hence

$$J(\pi^E) - J(\pi) \le \mathbb{E}\big[H - \tau\big] = \sum_{t=0}^{H-1} (H - t)\ \Pr(\tau = t) \le H \sum_{t=0}^{H-1} \Pr(\tau = t).$$

The trajectories separate at time t only if they were together until t, in which case the current state is the expert's state s_t^E, and the learner then disagrees. Writing $e(s) = \pi(a \neq \pi^E(s) \mid s)$,

$$\Pr(\tau = t) = \mathbb{E}\big[\mathbb{1}\{\tau \ge t\}\, e(s_t^E)\big] \le \mathbb{E}\big[e(s_t^E)\big] = \mathbb{E}_{s \sim d_t^E}[e(s)].$$

Summing over t gives $\sum_t \Pr(\tau = t) \le \epsilon H$, and the bound follows.

Lower bound. Take two states, good and bad. In good every action earns 1; the expert's action stays in good, the other action moves to bad, which is absorbing and earns 0. The expert never leaves good and earns H. A learner that takes the wrong action in good with probability ϵ has error ϵ on the expert's states, is still in good at time t with probability $(1 - \epsilon)^t$, and earns $\sum_{t=0}^{H-1}(1 - \epsilon)^t = (1 - (1 - \epsilon)^H)/\epsilon$. Using $(1 - \epsilon)^H \ge 1 - \epsilon H + \binom{H}{2}\epsilon^2 - \binom{H}{3}\epsilon^3$, the loss is at least $\binom{H}{2}\epsilon - \binom{H}{3}\epsilon^2 \ge \frac{1}{2}\epsilon H(H - 1)(1 - \epsilon H)$. ∎

Compare this with supervised learning. If the learner were evaluated on the expert's states, as a classifier is evaluated on its training distribution, the expected number of mistakes per episode would be ϵH and the loss at most ϵH. The extra factor H in the theorem comes from distribution shift: a mistake at time t can spoil all the remaining $H - t$ steps, because the learner has left the region where it was trained. On the windy lane the loss of the cloned policy grows like H^2, as the right panel of Figure 18.2 shows: from $H = 5$ to $H = 80$ (16 times longer) it grows from 0.064 to 17.1, about 270 times, a slope of 2.0 on the log-log plot.

PITFALL

More demonstrations do not cure distribution shift. They lower ϵ, so the cloned policy makes fewer mistakes, but they never show it the states it reaches after a mistake, because the expert never goes there. On the windy lane no amount of expert data ever contains $y = 2$. The remedies are to collect data in the learner's states (Section 18.3), to perturb the expert during data collection so that its demonstrations include recoveries (injecting noise into the expert's controls, as in DART (Laskey et al. 2017)), or to change the objective (Section 18.6).

Two more failure modes of behavior cloning are worth knowing. With *multimodal* demonstrations, such as passing an obstacle on the left half of the time and on the right the other half, a policy that predicts the average action with a squared loss drives straight into the obstacle; expressive policy classes (Section 18.8) address this. And with rich observations, a cloned policy can latch onto features that are correlated with the expert's actions without causing them, such as a brake light on the dashboard that turns on whenever the expert brakes; adding information

to the observation can then make imitation *worse*, a phenomenon called causal confusion (Haan et al. 2019).

18.3 Interactive imitation: DAgger

If the problem is that the learner visits states the expert never demonstrated, the obvious remedy is to ask the expert what to do in those states. *DAgger* (dataset aggregation) (Ross et al. 2011) does exactly this, iteratively: run the current learner, record the states it visits, ask the expert to label them with the actions it would have taken, add the labeled states to the data set, and retrain.

Algorithm 18.2 DAgger

Input: an expert that can be queried for $\pi^E(s)$ in any state; a supervised learner; number of iterations N; mixing weights $\beta_1, \dots, \beta_N \in [0,1]$ (for example $\beta_1 = 1$ and $\beta_i = 0$ afterwards).

1. Initialize the data set $\mathcal{D} \leftarrow \varnothing$ and any policy $\hat{\pi}_1$.
2. For $i = 1, \dots, N$:
   1. Let $\pi_i = \beta_i \pi^E + (1 - \beta_i)\hat{\pi}_i$ (at each step, follow the expert with probability β_i).
   2. Run π_i for some episodes and record the visited states s.
   3. Label each visited state with the expert's action: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, \pi^E(s))\}$.
   4. Train $\hat{\pi}_{i+1}$ by behavior cloning on all of $\mathcal{D}$.

Output: the last policy $\hat{\pi}_{N+1}$, or the best $\hat{\pi}_i$ on validation episodes.

The key step is 2.3: the labels come from the expert, but the states come from the learner. The first iteration, with $\beta_1 = 1$, is plain behavior cloning; each later iteration adds exactly the states that the current learner gets wrong. On the windy lane, the learner's own rollouts reach $y = 2, 3, \dots$, the expert labels them left, and after a few iterations the learner steers back to the center (Figure 18.1, right).

Figure 18.2 compares the two methods with equal numbers of expert labels. With one labeled episode they coincide (DAgger's first iteration is behavior cloning). With 10, DAgger earns 45.9 and behavior cloning 40.5; behavior cloning needs about 30 episodes to reach 44.4 and is still at 46.3 with 100, while DAgger reaches 47.0 with 30.

18.3.1 Why DAgger works ●●

DAgger's guarantee rests on two observations. The first is a finite-horizon version of the performance difference lemma (Lemma 3.10). Let $V_t^E(s)$ and $Q_t^E(s,a)$ be the expert's value and action value with $H - t$ steps to go. For any policy π with state distributions d_t^π,

$$J(\pi^E) - J(\pi) = \sum_{t=0}^{H-1} \mathbb{E}_{s \sim d_t^\pi}\Big[V_t^E(s) - \sum_a \pi(a \mid s)\, Q_t^E(s,a) \Big]. \tag{18.1}$$

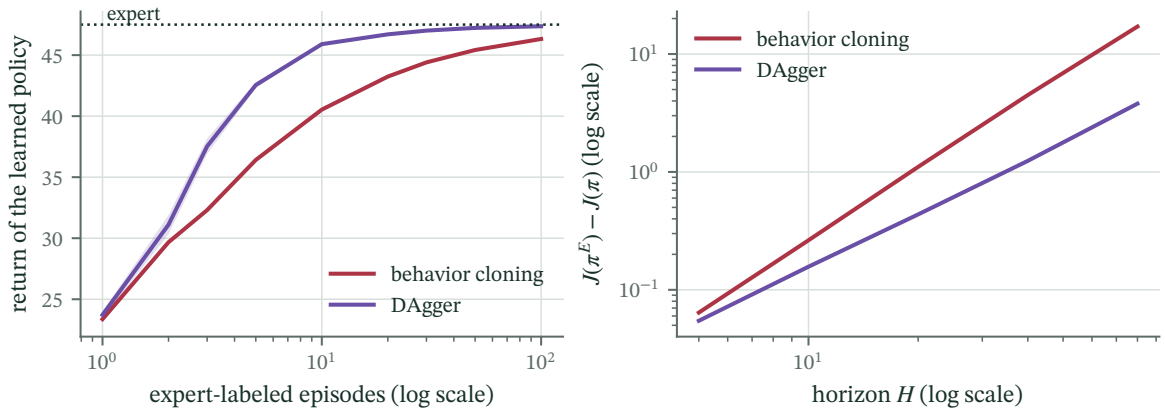


Figure 18.2. Behavior cloning against DAgger on the windy lane (20 runs; bands are ±1 standard error). Left: the return of the learned policy against the number of expert-labeled episodes (for DAgger, one expert episode followed by learner episodes labeled by the expert). DAgger with 10 labeled episodes (45.9) beats behavior cloning with 30 (44.4). Right: the loss against the expert as a function of the horizon, for policies trained on 10 labeled episodes. Behavior cloning's loss grows like H^2 (slope 2.0 on the log-log scale); DAgger's grows much more slowly (slope about 1.5; some of its mistakes still blow the car off the road, and those cannot be recovered).

(It follows by telescoping: $\mathbb{E}_\pi\big[\sum_t (r_t + V^E_{t+1}(s_{t+1}) - V^E_t(s_t))\big] = J(\pi) - J(\pi^E)$, with $V^E_H = 0$, and the conditional expectation of each term given s_t is $\sum_a \pi(a \mid s_t) Q^E_t(s_t, a) - V^E_t(s_t)$.) The expectation is over the *learner's* states. If the expert is *u-recoverable*, meaning that one wrong action followed by the expert costs at most u,

$$V^E_t(s) - Q^E_t(s, a) \le u \quad \text{for all } t, s, a,$$

then each term of (18.1) is at most u times the probability that π disagrees with the expert in s, and

$$J(\pi^E) - J(\pi) \le u\, H\, \mathbb{E}_{s \sim d_\pi}\big[\ell(s, \pi)\big], \qquad \ell(s, \pi) = \pi(a \neq \pi^E(s) \mid s),$$

where $d_\pi = \frac{1}{H}\sum_t d^\pi_t$ is the learner's average state distribution. What matters is the learner's error *on its own distribution*.

The second observation is that DAgger is an *online learning* algorithm for exactly this error. At iteration i, the learner commits to $\hat{\pi}_i$, and the environment reveals the loss $\ell_i(\pi) = \mathbb{E}_{s \sim d_{\hat{\pi}_i}}[\ell(s, \pi)]$, the error of any policy π on the states visited at iteration i. Training on the aggregate of all data so far is *follow-the-leader* on these losses, and with a strongly convex surrogate loss (or a regularized learner) it has *no regret*: its average loss approaches that of the best fixed policy in hindsight.

Theorem 18.2 DAgger's guarantee

Run DAgger with $\beta_1 = 1$ and $\beta_i = 0$ for $i \ge 2$, and suppose each iteration observes the exact loss ℓ_i (infinitely many samples). Suppose the learner has average regret at most γ_N over the N iterations, $\frac{1}{N}\sum_i \ell_i(\hat{\pi}_i) \le \epsilon_N + \gamma_N$, where $\epsilon_N = \min_{\pi \in \Pi} \frac{1}{N}\sum_i \ell_i(\pi)$ is the error of the best policy

in the class on the states of all iterations. Then some iterate $\hat{\pi} \in \{\hat{\pi}_1, \dots, \hat{\pi}_N\}$ satisfies

$$\mathbb{E}_{s \sim d_{\hat{\pi}}}[\ell(s, \hat{\pi})] \le \epsilon_N + \gamma_N + \frac{1}{N},$$

and if the expert is u-recoverable, $J(\pi^E) - J(\hat{\pi}) \le uH\big(\epsilon_N + \gamma_N + \frac{1}{N}\big)$.

Proof. For $i \ge 2$ the executed policy is $\hat{\pi}_i$ itself, so $\ell_i(\hat{\pi}_i) = \mathbb{E}_{s \sim d_{\hat{\pi}_i}}[\ell(s, \hat{\pi}_i)]$. For $i = 1$ the expert was executed, and we only use $\mathbb{E}_{s \sim d_{\hat{\pi}_1}}[\ell(s, \hat{\pi}_1)] \le 1 \le \ell_1(\hat{\pi}_1) + 1$. Summing over i and dividing by N,

$$\min_i \mathbb{E}_{s \sim d_{\hat{\pi}_i}}[\ell(s, \hat{\pi}_i)] \le \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}_{s \sim d_{\hat{\pi}_i}}[\ell(s, \hat{\pi}_i)] \le \frac{1}{N}\sum_{i=1}^{N} \ell_i(\hat{\pi}_i) + \frac{1}{N} \le \epsilon_N + \gamma_N + \frac{1}{N}.$$

The second claim follows from the bound $J(\pi^E) - J(\pi) \le uH\,\mathbb{E}_{d_\pi}[\ell(s, \pi)]$ derived above. ∎

Read together with Theorem 18.1, the theorem says that DAgger replaces the factor H^2 by uH. If mistakes are recoverable at a cost that does not grow with the horizon (u a constant), DAgger's loss is linear in H, like supervised learning. If mistakes are irrecoverable, as in the good-and-bad example of the proof above, u is of order H and DAgger gains nothing: no algorithm can undo an irreversible error. The quantity ϵ_N is also different from behavior cloning's ϵ: it is the error of the best policy in the class on the *learner's* states, which can be larger, because those states may be harder. Ross et al. (2011) also give versions with finitely many samples per iteration, and the earlier reductions of Ross and Bagnell (2010) achieve linear dependence on H with non-stationary policies trained one time step at a time.

DAgger's assumption is strong: an expert that can be queried in any state the learner reaches, including states the expert would never get into. A human driving instructor can say "steer left" when shown a car drifting off the road, but labeling thousands of states without being in control of the car is tiring and unnatural, and querying an expert in dangerous states can be unsafe. Practical variants let the human take over only when the learner is about to fail and record the corrections, and DAgger works best when the expert is a program, such as a planner with access to information the learner lacks.

CHECKPOINT

1. On the windy lane, why can 1,000 expert demonstrations not teach behavior cloning what to do at $y = 3$?
2. What is ϵH for the cloned policy of Section 18.2, and why is its actual loss several times larger?
3. In which kind of problem does DAgger have no advantage over behavior cloning?

Answers.

1. The expert never visits $y = 3$, so no demonstration contains it; more data from the same expert only repeats $y = 0$ and $y = 1$.
2. $\epsilon H = 0.032 \times 50 = 1.6$. Each mistake can move the car to unfamiliar positions where the policy acts at random and keeps losing reward, possibly for the rest of the episode.
3. In problems where a single mistake is irrecoverable (the recoverability constant u is of order H), such as falling into an absorbing bad state.

18.4 Inverse reinforcement learning

Imitation copies behavior. *Inverse reinforcement learning* asks for more: find a reward function r for which the expert's behavior is optimal, or nearly so (Ng and Russell 2000). A reward is a far more compact description of a task than a policy, it can be inspected, and it transfers: the reward "reach the goal, avoid the water" applies to a new map where the old policy does not. But inferring it is harder than it looks.

The problem is ill-posed. Many rewards explain the same behavior. Three kinds of ambiguity are always present.

- *Degenerate rewards.* With $r \equiv 0$ every policy is optimal, the expert's included. Any useful formulation must exclude such trivial explanations, for example by requiring the expert to be better than the alternatives by a margin.
- *Scaling and shifting.* Multiplying a reward by a positive constant does not change which policies are optimal, and neither does adding a constant to every reward in a problem where all episodes have the same length.
- *Reward shaping.* For any function Φ on states, the reward

  $$r'(s, a, s') = r(s, a, s') + \gamma\, \Phi(s') - \Phi(s)$$

  has exactly the same optimal policies as r (Ng et al. 1999) (Exercise 18.6). Shaping can change rewards drastically, for example turning a sparse reward into a dense one, without changing optimal behavior, so behavior alone can never distinguish r from r'.

Beyond these, a deterministic expert reveals only which action it prefers in each state, not by how much. On the Trail (Section 1.3), an expert that always goes all the way tells us only that $\gamma\, r_{\text{finish}} > r_{\text{exit}}(\text{Middle})$ and $\gamma \max\{r_{\text{exit}}(\text{Middle}), \gamma r_{\text{finish}}\} > r_{\text{exit}}(\text{Start})$. With $\gamma = 0.9$, the book's rewards $(1, 2, 4)$ satisfy both inequalities, and so do $(0, 0, 1)$, $(3, 3, 5)$ and infinitely many others.

Feature matching. A productive way around the ambiguity is to stop asking for *the* reward and ask instead for a policy that does as well as the expert under *every* plausible reward. Suppose the reward is linear in known features, $r_w(s) = w^\top \phi(s)$ with $\|w\|_2 \le 1$, and define the *feature expectations* of a policy, $\mu(\pi) = \mathbb{E}_\pi\big[\sum_t \gamma^t \phi(s_t)\big]$. Then $J_w(\pi) = w^\top \mu(\pi)$ for every w, and by the Cauchy–Schwarz inequality,

$$|J_w(\pi) - J_w(\pi^E)| = |w^\top(\mu(\pi) - \mu(\pi^E))| \le \|\mu(\pi) - \mu(\pi^E)\|_2.$$

A policy whose feature expectations match the expert's to within ε is within ε of the expert under every reward in the class, whichever one the expert had in mind. *Apprenticeship learning* (Abbeel and Ng 2004) finds such a policy by alternating between finding the reward that best separates the expert's feature expectations from those of the policies found so far, and computing an optimal policy for that reward. *Maximum-margin planning* (Ratliff et al. 2006) casts the same idea as a structured large-margin problem: find a reward under which the expert's behavior beats every other behavior by a margin that grows with how different the other behavior is.

Feature matching leaves one question open. Many policies, and mixtures of policies, match the expert's features; which one should we choose? The next section answers with a principle that also resolves the remaining ambiguity of the reward.

KEY IDEA

Behavior underdetermines reward. Every IRL method makes the problem well-posed by adding something: a margin, a class of features, a prior, or, in the next section, the principle of maximum entropy.

18.5 Maximum-entropy IRL

Among all the distributions over trajectories that match the expert's feature expectations, choose the one that is otherwise as uncertain as possible: the one with the largest entropy. This is the *principle of maximum entropy* (Jaynes 1957), and Ziebart et al. (2008) applied it to IRL. For a problem with deterministic dynamics, the maximum-entropy distribution subject to matching feature expectations has the exponential-family form

$$p_\psi(\tau) = \frac{\exp\big(r_\psi(\tau)\big)}{Z_\psi}, \qquad r_\psi(\tau) = \sum_{t=0}^{H-1} r_\psi(s_t) = \psi^\top \Phi(\tau), \qquad Z_\psi = \sum_\tau \exp\big(r_\psi(\tau)\big), \tag{18.2}$$

where $\Phi(\tau) = \sum_t \phi(s_t)$ is the trajectory's feature count, ψ the reward weights, and the sum in Z_ψ runs over all trajectories from the start state. In words: trajectories with higher reward are exponentially more likely, and trajectories with equal reward are equally likely. The demonstrator is modeled as *noisily* optimal, which is more realistic than exact optimality and has a crucial technical benefit: it makes the likelihood of the demonstrations a smooth function of ψ.

The likelihood and its gradient. Fit ψ by maximum likelihood on the demonstrations $\tau_1, \dots, \tau_n$:

$$\mathcal{L}(\psi) = \frac{1}{n}\sum_{i=1}^{n} \log p_\psi(\tau_i) = \psi^\top \bar{\Phi}_E - \log Z_\psi, \qquad \bar{\Phi}_E = \frac{1}{n}\sum_i \Phi(\tau_i).$$

Differentiating $\log Z_\psi$ gives $\nabla_\psi \log Z_\psi = \sum_\tau \frac{\exp(\psi^\top \Phi(\tau))}{Z_\psi}\Phi(\tau) = \mathbb{E}_{\tau\sim p_\psi}[\Phi(\tau)]$, so

$$\nabla_\psi \mathcal{L}(\psi) = \bar{\Phi}_E - \mathbb{E}_{\tau\sim p_\psi}\big[\Phi(\tau)\big]. \tag{18.3}$$

The gradient is the difference between the demonstrated feature counts and those the model expects. At the maximum the two match: maximum-likelihood fitting *is* feature matching, and the fitted distribution is the maximum-entropy one among those that match. The objective is concave (its Hessian is minus the covariance of Φ under p_ψ), so gradient ascent finds the global maximum.

Computing the gradient. The sum over all trajectories in Z_ψ and in the expected features is exponentially large, but dynamic programming computes both. A backward pass of *soft value iteration* computes

$$Q_t(s,a) = r_\psi(s) + \sum_{s'} P(s' \mid s,a)\, V_{t+1}(s'), \qquad V_t(s) = \log \sum_a e^{Q_t(s,a)}, \qquad V_H = 0,$$

which is the Bellman recursion with the maximum replaced by its soft version, the log-sum-exp. For deterministic dynamics $V_0(s_0) = \log Z_\psi$, and the trajectory distribution (18.2) is generated

by the policy $\pi_t(a \mid s) = \exp\big(Q_t(s,a) - V_t(s)\big)$, the soft-optimal policy of maximum-entropy RL (Section 13.5). A forward pass then propagates the state distribution under this policy and accumulates the expected feature counts.

Algorithm 18.3 Maximum-entropy IRL

Input: demonstrations $\tau_1, \dots, \tau_n$; a model P; state features $\phi(s)$; horizon H; step size η.

1. Compute the demonstrated feature counts $\bar{\Phi}_E = \frac{1}{n}\sum_i \sum_t \phi(s_t^i)$, and initialize ψ.
2. Repeat until the gradient is small:
   1. *Backward pass:* soft value iteration for the reward $r_\psi(s) = \psi^\top \phi(s)$ gives V_t, Q_t and the policy $\pi_t(a \mid s) = \exp(Q_t(s,a) - V_t(s))$.
   2. *Forward pass:* starting from the demonstrations' initial states, propagate $D_{t+1}(s') = \sum_{s,a} D_t(s)\,\pi_t(a \mid s)\,P(s' \mid s,a)$ and compute $\bar{\Phi}_\psi = \sum_t \sum_s D_t(s)\,\phi(s)$.
   3. $\psi \leftarrow \psi + \eta\,(\bar{\Phi}_E - \bar{\Phi}_\psi)$.

Output: the reward weights ψ and the soft-optimal policy π.

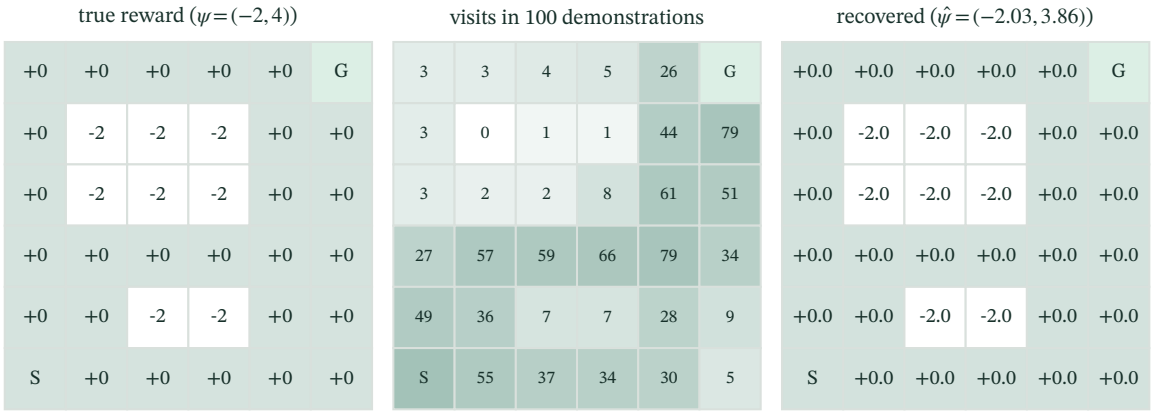


Figure 18.3. Maximum-entropy IRL on a grid. Left: the true reward, 0 on plain cells, −2 on water and +4 on the goal (top right); the start is at the bottom left. Middle: how often 100 demonstrations of 16 steps, sampled from the soft-optimal policy for the true reward, visit each cell. Right: the reward recovered from those demonstrations, $\hat{\psi} = (-2.03, 3.86)$. The demonstrations rarely touch the water, so its weight rests on few observations.

Figure 18.3 runs the algorithm on a 6×6 grid with five actions (four moves and stay) and deterministic dynamics. Each cell has a type, and the features are indicators of the two non-plain types, water and goal, so the reward of a cell is ψ_{water}, ψ_{goal} or 0. From 100 demonstrations generated with $\psi = (-2, 4)$, maximum-entropy IRL recovers $\hat{\psi} = (-2.03, 3.86)$. At the solution the model's expected feature counts match the demonstrated ones, 0.28 water visits and 5.85 goal visits per episode. Dropping the indicator of plain cells was not an accident: since every cell has exactly one type, adding the same constant c to all three weights adds cH to every trajectory's

reward and changes nothing (Exercise 18.9), so one weight must be fixed.

Stochastic dynamics. With random transitions, (18.2) is not the right model: reweighting whole trajectories by their reward would let the demonstrator influence the outcomes of coin flips it does not control. The correct generalization, the principle of *maximum causal entropy*, maximizes the entropy of the actions given the past and leads to exactly the soft value iteration above, with the expectation over s' in the backward pass (Ziebart 2010). Its log-likelihood is $\sum_{i,t} \log \pi_t(a_t^i \mid s_t^i)$, and its gradient is again the feature-count difference (in expectation over the dynamics). This is the version implemented in the lab.

Beyond small models. Algorithm 18.3 needs a model and a full soft value iteration per gradient step. *Guided cost learning* (Finn et al. 2016) removes both requirements: it represents the reward by a neural network, estimates the partition function by importance sampling with trajectories from a policy that is itself trained, by reinforcement learning, to be a good sampler for the current reward. This structure, a reward updated to distinguish expert trajectories from the policy's, and a policy updated to score well under the reward, leads directly to the adversarial methods of the next section.

18.6 Adversarial imitation

Maximum-entropy IRL learns a reward and then a policy for it. If the goal is only the policy, the reward can be treated as an intermediate quantity and optimized away. The *occupancy measure* of a policy, $\rho_\pi(s,a)$, is its distribution over state-action pairs (in a finite-horizon problem, $\frac{1}{H}\sum_t \Pr_\pi(s_t = s, a_t = a)$; in a discounted one, the visitation distribution of Definition 3.8). Two policies with the same occupancy measure have the same expected return under every reward, and for stationary policies in a discounted problem the occupancy measure determines the policy, through $\pi(a \mid s) = \rho_\pi(s,a)/\sum_{a'} \rho_\pi(s,a')$. Imitation can therefore be posed as *making* ρ_π *equal to* ρ_E.

Ho and Ermon (2016) showed that a family of IRL formulations followed by RL is equivalent to minimizing a divergence between ρ_π and ρ_E, and that a particular choice of regularizer gives the Jensen–Shannon divergence, which can be minimized with the machinery of generative adversarial networks (Goodfellow et al. 2014). A *discriminator* $D(s,a) \in (0,1)$ learns to tell expert pairs from the learner's pairs, and the learner is rewarded for fooling it. With the convention that D is the probability that a pair came from the expert, the *generative adversarial imitation learning* (GAIL) objective is

$$\min_\pi \max_D \; \mathbb{E}_{(s,a)\sim\rho_E}\big[\log D(s,a)\big] + \mathbb{E}_{(s,a)\sim\rho_\pi}\big[\log(1 - D(s,a))\big] - \lambda\,\mathcal{H}(\pi), \tag{18.4}$$

where $\mathcal{H}(\pi)$ is the policy's (causal) entropy and $\lambda \geq 0$. (The original paper uses the opposite labels, with D the probability of the learner; the objective and the reward then change signs consistently.)

For a fixed policy the inner maximization has a closed form. The maximizing discriminator is $D^*(s,a) = \rho_E(s,a)/(\rho_E(s,a) + \rho_\pi(s,a))$, and substituting it gives

$$\mathbb{E}_{\rho_E}\Big[\log \frac{\rho_E}{\rho_E + \rho_\pi}\Big] + \mathbb{E}_{\rho_\pi}\Big[\log \frac{\rho_\pi}{\rho_E + \rho_\pi}\Big] = 2\,\mathrm{JS}(\rho_E, \rho_\pi) - \log 4,$$

where $\mathrm{JS}(p,q) = \frac{1}{2}D_{\mathrm{KL}}(p\|m) + \frac{1}{2}D_{\mathrm{KL}}(q\|m)$ with $m = \frac{1}{2}(p+q)$ is the Jensen–Shannon divergence (Exercise 18.10). So GAIL minimizes the Jensen–Shannon divergence between the occupancy measures, with an entropy bonus. In practice the two players are trained in alternation: the discriminator by logistic regression, the policy by a policy-gradient method such as TRPO or PPO (Algorithm 12.2) with the reward $-\log(1 - D(s,a))$ or $\log D(s,a) - \log(1 - D(s,a))$.

Algorithm 18.4 Generative adversarial imitation learning (GAIL)

Input: expert state-action pairs from ρ_E; a policy π_θ; a discriminator D_w; entropy weight λ.

1. Repeat for the chosen number of iterations:
   1. Run π_θ in the environment and collect state-action pairs from ρ_{π_θ}.
   2. Discriminator step: increase $\mathbb{E}_{\rho_E}[\log D_w(s,a)] + \mathbb{E}_{\rho_{\pi_\theta}}[\log(1 - D_w(s,a))]$ by a gradient step on w, using expert pairs and the new learner pairs.
   3. Policy step: one or a few policy-gradient updates (for example PPO) on the reward $r(s,a) = \log D_w(s,a) - \log(1 - D_w(s,a))$, with an entropy bonus λ.

Output: the policy π_θ.

Note what GAIL needs and what it does not. It never queries the expert beyond the demonstrations, but it does interact with the environment: the learner's own pairs in step 1.1 come from running it. That interaction is what removes the compounding errors of behavior cloning. The learner is trained on its own state distribution, and a state that the expert never visits is penalized by the discriminator because it makes the learner's occupancy differ from the expert's. In the lab, a tabular version with exact occupancies drives the Jensen–Shannon divergence on the grid of Figure 18.3 from 0.45 to below 10^{-5} in 300 iterations. The price is the instability of adversarial training, sensitivity to the discriminator's capacity and regularization, and a learned reward that is not meant to be meaningful on its own: at the optimum D is $\frac{1}{2}$ everywhere and the reward is constant.

18.6.1 Recovering rewards adversarially ●●

Adversarial inverse reinforcement learning (AIRL) (Fu et al. 2018) restructures the discriminator so that a reward can be read off. It uses

$$D(s,a,s') = \frac{\exp f(s,a,s')}{\exp f(s,a,s') + \pi(a \mid s)}, \qquad f(s,a,s') = g(s) + \gamma h(s') - h(s),$$

for which the policy's reward $\log D - \log(1-D) = f(s,a,s') - \log \pi(a \mid s)$ is an entropy-regularized reward f. The form of f is a reward g plus a potential-based shaping term in h, which is exactly the ambiguity of Section 18.4; separating the two lets g absorb the reward and h the shaping. Fu et al. (2018) show that, for deterministic dynamics and a true reward that depends only on the state, g recovers the reward up to a constant, and they demonstrate that the recovered rewards transfer to environments with changed dynamics, where a learned policy would not.

18.6.2 Imitation as moment matching ●●●

Swamy et al. (2021) unify these methods as *moment matching*: choose a class of functions (rewards or action values) and make the learner's moments of those functions equal to the expert's. Behavior cloning matches moments of the expert's action-value functions on the expert's own states, and inherits the $O(\epsilon H^2)$ gap of Theorem 18.1. Methods that match reward moments on the learner's own state distribution, as GAIL does, achieve an $O(\epsilon H)$ gap, at the price of interacting with the environment but not with the expert. Interactive methods such as DAgger match moments on the learner's states with the expert's help, and achieve $O(\epsilon H)$ under a recoverability condition. The framework clarifies when each kind of information (demonstrations, a simulator, an interactive expert) buys what, and it is a good starting point for research on the sample complexity of imitation.

18.7 Learning rewards from preferences

Demonstrations require someone who can perform the task. For many tasks, people cannot perform them well but can judge them: few of us can perform a backflip, but anyone can tell a good one from a bad one in a short video clip, and a reader who cannot write a perfect summary can say which of two summaries is better. *Preference learning* fits a reward to such judgments.

The Bradley–Terry model. Let τ^1 and τ^2 be two items to be compared: two trajectories, two short trajectory segments, or, for a language model, two responses to the same prompt. Each item has a scalar score $R(\tau)$; for a trajectory segment it is the sum of the rewards along it, $R(\tau) = \sum_t r(s_t, a_t)$. The Bradley–Terry model (Bradley and Terry 1952) says that a labeler prefers τ^1 to τ^2 with probability

$$\Pr\big(\tau^1 \succ \tau^2\big) = \frac{\exp R(\tau^1)}{\exp R(\tau^1) + \exp R(\tau^2)} = \sigma\big(R(\tau^1) - R(\tau^2)\big), \tag{18.5}$$

where $\sigma(z) = 1/(1+e^{-z})$ is the logistic function. In words: the item with the larger score is preferred more often, the probability depends only on the difference of the scores, and a difference of 0 gives a coin flip. Nothing in (18.5) depends on the items being trajectories; Chapter 22 uses it with responses y_1, y_2 to a prompt x and a reward model $r_\psi(x, y)$ (Section 22.2).

Fitting a reward. Given comparisons $(\tau_i^1, \tau_i^2, y_i)_{i=1}^m$, where $y_i = 1$ if the labeler preferred τ_i^1 and 0 otherwise, and a parametric reward r_ψ, maximize the log-likelihood

$$\mathcal{L}(\psi) = \sum_{i=1}^{m} \Big[y_i \log \sigma(\Delta_i) + (1 - y_i) \log \sigma(-\Delta_i) \Big], \qquad \Delta_i = R_\psi(\tau_i^1) - R_\psi(\tau_i^2).$$

This is logistic regression on score differences. Its gradient is $\nabla_\psi \mathcal{L} = \sum_i \big(y_i - \sigma(\Delta_i)\big) \nabla_\psi \Delta_i$: each comparison pushes the score difference towards the observed preference, in proportion to how surprised the model was. For a reward linear in features, $\Delta_i = \psi^\top(\Phi(\tau_i^1) - \Phi(\tau_i^2))$ and the objective is concave.

What is identified. Because (18.5) depends only on differences of scores, adding a constant c to every per-step reward adds cL to the score of every segment of length L and leaves every preference probability unchanged when the compared segments have equal lengths. The reward is

therefore identified at best up to an additive constant. Its scale, on the other hand, is fixed by the model: the assumption that preference noise follows the logistic function with unit scale. A labeler who is more consistent than the model assumes would make the fitted rewards larger. And comparisons only constrain the reward along the segments that are compared, which is the coverage problem of Chapter 17 in a new guise.

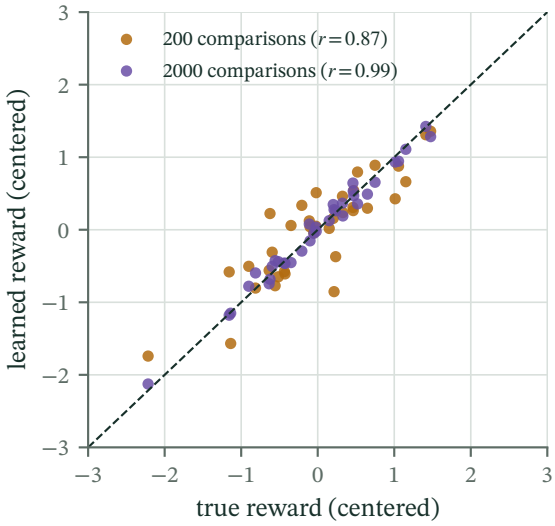


Figure 18.4. A reward learned from pairwise comparisons against the true reward on the 6×6 grid, one parameter per cell (both centered to mean zero, since the constant is not identified). Comparisons are between random six-step segments, labeled by a simulated person who follows the Bradley–Terry model with the true reward. With 200 comparisons the learned reward correlates 0.87 with the truth; with 2,000, 0.99.

Figure 18.4 shows the method at work on the grid, now with a hidden reward drawn at random for every cell and a learned reward with one parameter per cell. The correlation between learned and true rewards rises from 0.87 with 200 comparisons to 0.99 with 2,000. A high correlation is not the whole story, though. In the lab, planning greedily with a reward learned from 1,000 comparisons (correlation 0.988) achieves the best possible true reward, while a reward learned from 3,000 comparisons, with a slightly *higher* correlation (0.992), leads to a plan that is worse under the true reward: the planner seeks out the cells that the learned reward overrates, so errors at the top of the reward matter more than the average error. This is the first appearance of *reward hacking*, the exploitation of errors in a learned reward, which Chapter 22 studies at scale (Section 22.8).

Preferences for deep RL. Christiano et al. (2017) turned these ideas into a practical method for deep reinforcement learning. A reward network is trained on human comparisons of pairs of short video clips of the agent, while a policy is trained by RL on the current reward network, and the clips to be compared are chosen where an ensemble of reward networks disagrees most. Agents learned Atari games and simulated robot behaviors from comparisons alone, including a backflip that would have been hard to specify with a hand-written reward, from about 900 human judgments. Two lessons from this work carry over to language models. The reward model must keep being trained on the current policy's behavior: a policy optimized against a

frozen learned reward eventually finds behaviors that the reward model scores highly but people do not like (Ibarz et al. 2018). And the policy should not be optimized too hard against a proxy: in RLHF, a KL penalty towards a reference policy limits how far the policy can move from the region where the reward model is trustworthy (Section 22.2).

LAB

Lab 18.1: Imitation, inverse RL and preferences (`labs/ch18_imitation.py`, about 45 minutes; runs in a few seconds). You compute the expert of the windy lane by value iteration, clone it and measure how its errors compound, run DAgger and compare the growth of the loss with the horizon, check the maximum-entropy IRL gradient against finite differences and recover the grid's reward, learn a reward from Bradley–Terry comparisons and plan with it, and watch a tabular GAIL match the expert's occupancy measure. The library code is in `rlcourse/imitation.py`.

CHECKPOINT

1. Under the Bradley–Terry model, what is the probability that a segment with return 3 is preferred to one with return 1?
2. Why can preference learning identify a reward only up to an additive constant?
3. What does the GAIL discriminator converge to when the learner matches the expert's occupancy measure?

Answers.

1. $\sigma(2) = 1/(1 + e^{-2}) \approx 0.88$.
2. Preference probabilities depend only on differences of returns, and a constant added to every step's reward cancels in the difference of two segments of equal length.
3. $D = \rho_E/(\rho_E + \rho_\pi) = \frac{1}{2}$ everywhere, so its reward is constant and carries no information.

18.8 Modern imitation ●●

Since about 2021, imitation learning has become one of the main ways of training robot manipulation policies, largely because of better policy classes and much larger data sets.

Expressive policies. Human demonstrations are often multimodal (there are several good ways to grasp a cup) and discontinuous (a small change in the state flips the right action). A Gaussian policy fitted by maximum likelihood averages the modes. *Implicit behavior cloning* (Florence et al. 2022) instead learns an energy function $E_\theta(s, a)$ and acts by minimizing it over actions, which represents multiple modes and sharp boundaries naturally. *Diffusion policies* (Chi et al. 2023) represent the action distribution by a denoising diffusion model conditioned on the observation, which generates samples from complex distributions by iteratively refining random noise; they predict a short sequence of future actions at a time and were reported to improve substantially on earlier behavior-cloning methods on a range of manipulation tasks. *Action chunking* with transformers (Zhao et al. 2023) also predicts sequences of actions, and the authors argue that acting in chunks reduces the effective horizon over which errors compound, a direct response to Theorem 18.1.

Scale. Large data sets of real robot demonstrations, collected by teleoperation over many months, have made it possible to train transformer policies that follow language instructions across many tasks (Brohan, Brown, Carbajal, Chebotar, and others 2023), and to initialize such policies from vision-language models trained on web data (Brohan, Brown, Carbajal, Chebotar, Chen, et al. 2023). These are behavior-cloning systems at heart, and their reported successes owe much to data coverage: diverse demonstrations, including imperfect ones with recoveries, reduce the distribution shift that Section 18.2 analyzed. How much interaction, correction or reinforcement learning such systems need on top of imitation is an open and active question (see Section 24.3).

18.8.1 Imitation through soft Q-learning ●●●

Two methods avoid adversarial training by working directly with soft action values (the maximum-entropy RL of Section 13.5). *SQIL* (Reddy et al. 2020) runs soft Q-learning on a replay buffer in which every demonstrated transition has reward 1 and every transition of the agent has reward 0; the agent is thereby encouraged to return to demonstrated states and to act as the expert did there, which addresses distribution shift with a simple change to standard RL. *IQ-Learn* (Garg et al. 2021) learns a single soft Q-function from which both the policy and an implicit reward follow, $r(s,a) = Q(s,a) - \gamma\, \mathbb{E}_{s'}[V(s')]$ with V the soft value, and shows that the adversarial IRL problem can be rewritten as an optimization over the Q-function alone. Both report strong results with few demonstrations.

RESEARCH NOTE

Open questions: How should demonstrations, interventions, preferences and rewards be combined in one learner? What are the sample-complexity limits of imitation from observations without actions? How can a policy trained by imitation know when it is out of its depth and ask for help? When are learned rewards trustworthy enough to optimize hard, and how can reward hacking be detected before deployment? Osa et al. (2018) survey the algorithmic landscape up to 2018.

HISTORICAL NOTE

Learning by imitation has a long history in robotics under names such as "programming by demonstration" and "learning from demonstration". ALVINN (Pomerleau 1989) is the classic early neural-network example. Inverse reinforcement learning was formulated in the late 1990s and given its first algorithms by Ng and Russell (2000), followed by apprenticeship learning (Abbeel and Ng 2004), maximum-margin planning (Ratliff et al. 2006) and maximum-entropy IRL (Ziebart et al. 2008). The reduction of imitation to no-regret online learning, and DAgger, are due to Ross et al. (2011). GAIL (Ho and Ermon 2016) connected IRL to generative adversarial networks, and learning from human preferences (Christiano et al. 2017) became the basis of RLHF.

18.9 Summary

- When a reward is hard to write but behavior is easy to show or to judge, we can learn from demonstrations, from an interactive expert, or from preferences.
- Behavior cloning is supervised learning of the expert's actions. It is simple and often strong, but its errors compound: a learner with error ϵ on the expert's states can lose $O(\epsilon H^2)$ over

a horizon H, because each mistake leads to states the demonstrations never covered. More demonstrations reduce ϵ but never show the learner how to recover.

- DAgger queries the expert in the states the learner visits and retrains on the aggregate; as a no-regret online learner it achieves a loss of order $u\epsilon H$, linear in the horizon when mistakes are recoverable at bounded cost u.
- Inverse reinforcement learning infers a reward that explains the behavior. It is ill-posed (trivial rewards, scaling, potential-based shaping), and every method adds a principle to resolve the ambiguity; feature matching guarantees good performance under every reward in a linear class.
- Maximum-entropy IRL models the expert as noisily optimal, $p(\tau) \propto \exp(r_\psi(\tau))$; its gradient is the difference between demonstrated and expected feature counts, computed with soft value iteration and a forward pass.
- GAIL matches occupancy measures by minimizing a Jensen–Shannon divergence with a discriminator as the reward; it avoids compounding errors by training on the learner's own states, at the cost of environment interaction and adversarial instability. (●●) AIRL structures the discriminator to recover a transferable reward.
- The Bradley–Terry model, $\Pr(\tau^1 \succ \tau^2) = \sigma(R(\tau^1) - R(\tau^2))$, turns pairwise comparisons into a logistic-regression problem for a reward, identified up to a constant. Optimizing a learned reward hard can exploit its errors.
- (●●) Modern robot imitation relies on expressive policy classes (energy-based, diffusion, action chunking) and large demonstration data sets.

The next chapter drops another assumption of the earlier chapters: that the agent can see the state.

18.10 Further Reading

Osa et al. (2018) is a thorough survey of imitation learning from the algorithmic point of view, covering behavior cloning, IRL and their use in robotics. For the analysis of behavior cloning and its compounding errors, read Ross and Bagnell (2010), and for DAgger and the reduction to online learning, Ross et al. (2011); both are short and clear.

On inverse reinforcement learning, Ng and Russell (2000) set up the problem and its ambiguities, and Ng et al. (1999) is the reference on potential-based shaping. Abbeel and Ng (2004) (apprenticeship learning by feature matching) and Ratliff et al. (2006) (maximum-margin planning) are the classic algorithms. Ziebart et al. (2008) introduced maximum-entropy IRL; the causal version for stochastic dynamics is developed in Ziebart (2010), and Levine (2018) explains its connection to control as inference. Finn et al. (2016) show how to scale it with neural-network rewards.

For adversarial imitation, read Ho and Ermon (2016) (GAIL, including the derivation from IRL with a regularizer) and Fu et al. (2018) (AIRL and reward transfer). Swamy et al. (2021) give a unifying game-theoretic view that clarifies what each kind of information buys.

On learning from preferences, Bradley and Terry (1952) is the original statistics paper; Christiano et al. (2017) is the paper that made preference-based RL practical, and Ibarz et al. (2018)

combines preferences with demonstrations and documents reward-model exploitation. Chapter 22 continues the story with language models.

(●●●) For modern imitation, start with Florence et al. (2022), Chi et al. (2023) and Zhao et al. (2023) on policy classes, and Brohan, Brown, Carbajal, Chebotar, and others (2023) on scale. Reddy et al. (2020) and Garg et al. (2021) show how imitation can be done with the tools of soft Q-learning.

18.11 Exercises

Exercise 18.1 ● Reading the windy lane

In Section 18.2, the cloned policy has error $\epsilon = 0.032$ on the expert's states over $H = 50$ steps and loses 7.0 against the expert. (a) Compute ϵH and ϵH^2, and place the actual loss between them. (b) Explain in words why the loss exceeds ϵH. (c) Without running anything, predict whether a symmetric wind (gusts to the left or right, each with probability 0.15) would make behavior cloning better or worse, and why.

Exercise 18.2 ●● When the bound is tight

(a) For the two-state good/bad example in the proof of Theorem 18.1, show that the loss is exactly $H - (1 - (1 - \epsilon)^H)/\epsilon$. (b) Evaluate it for $\epsilon = 0.01$ and $H = 10, 30, 100$ and compare with ϵH^2 and $\epsilon H^2/2$. (c) Explain why this example is the worst case for behavior cloning, and why DAgger cannot do better on it.

Exercise 18.3 ●● Refining the bound

(a) Show that the proof of Theorem 18.1 actually gives $J(\pi^E) - J(\pi) \le \sum_{t=0}^{H-1} (H - t)\,\epsilon_t$, where ϵ_t is the learner's error on d_t^E, and that this is at most $\epsilon H(H+1)/2$ when $\epsilon_t = \epsilon$ for every t. (b) Suppose the learner errs only in the last k steps of the episode. What does the refined bound give? Interpret.

Exercise 18.4 ●● Reading DAgger's bound

(a) Compute the recoverability constant u of the expert in the good/bad example, and evaluate the bound of Theorem 18.2 there. (b) For a lane in which a gust never pushes the car off the road from the positions the expert or the learner typically visits, argue that u is bounded by a constant independent of H. (c) The theorem bounds the error of the *best* iterate. Why does DAgger in practice often return the last one, and what could go wrong?

Exercise 18.5 ● Many rewards, one behavior

On the Trail with $\gamma = 0.9$, an expert always goes all the way. (a) Write the conditions on the rewards $(r_{\text{Start}}, r_{\text{Middle}}, r_{\text{finish}})$ of exit at Start, exit at Middle and finish under which this behavior is optimal. (b) Give three different reward vectors that satisfy them, one of which has $r_{\text{finish}} < r_{\text{Middle}}$. (c) Which of your rewards would make the hiker exit at Middle if the discount factor were 0.5? What does this say about transferring a learned reward to a new setting?

Exercise 18.6 ●● Shaping does not change optimal policies

Let $r'(s,a,s') = r(s,a,s') + \gamma\Phi(s') - \Phi(s)$ for a bounded function Φ with $\Phi = 0$ on terminal states. (a) Show that $Q'^*(s,a) = Q^*(s,a) - \Phi(s)$, and conclude that r and r' have the same optimal policies. (b) Show the analogous statement for the action values of any fixed policy. (c) On the Trail, choose Φ so that the shaped reward of go at Start is positive, and verify (a) numerically.

Exercise 18.7 ● Feature matching bounds the loss

(a) Prove that $|J_w(\pi) - J_w(\pi^E)| \le \|w\|_2 \, \|\mu(\pi) - \mu(\pi^E)\|_2$ for a reward $r_w(s) = w^\top \phi(s)$. (b) Show by an example that matching feature expectations does not require matching the expert's policy. (c) Why does the bound say nothing if the true reward is not linear in the chosen features?

Exercise 18.8 ●● The maximum-entropy gradient

(a) Derive (18.3) from (18.2). (b) Show that the Hessian of $\mathcal{L}$ is $-\operatorname{Cov}_{\tau\sim p_\psi}[\Phi(\tau)]$, so that $\mathcal{L}$ is concave. When is it strictly concave? (c) Prove that for deterministic dynamics $V_0(s_0) = \log Z_\psi$, by induction on the number of steps to go.

Exercise 18.9 ● A constant reward changes nothing

(a) Show that adding a constant c to $r_\psi(s)$ for every state increases $\log Z_\psi$ by cH and leaves $p_\psi(\tau)$ unchanged. (b) Explain why the grid features of Section 18.5 omit the indicator of plain cells, and what would happen to gradient ascent if it were included. (c) Is the same true in a problem where episodes can end early in a terminal state? Explain.

Exercise 18.10 ●● GAIL minimizes the Jensen–Shannon divergence

(a) For fixed occupancy measures, show that $D^*(s,a) = \rho_E/(\rho_E + \rho_\pi)$ maximizes the inner objective of (18.4), pointwise for each (s,a). (b) Substitute it and show that the value is $2\,\mathrm{JS}(\rho_E, \rho_\pi) - \log 4$. (c) Show that the reward $\log D^* - \log(1 - D^*)$ equals $\log(\rho_E/\rho_\pi)$, and explain why maximizing its expectation under ρ_π pushes the learner towards the expert's state-action pairs.

Exercise 18.11 ● The Bradley–Terry likelihood

(a) Derive the gradient of the Bradley–Terry log-likelihood for a linear reward and show that the objective is concave. (b) Two segments have feature counts $(2, 0)$ and $(1, 1)$, and the labeler preferred the first. With $\psi = (0, 0)$, what are the predicted probability and the gradient? After one gradient step of size 1, what is the new probability? (c) Show that adding a constant to every per-step reward leaves the likelihood unchanged for segments of equal length but not for segments of different lengths.

Exercise 18.12 ●● The scale of a learned reward

Suppose a labeler follows $\Pr(\tau^1 \succ \tau^2) = \sigma\big((R(\tau^1) - R(\tau^2))/T\big)$ for some unknown temperature $T > 0$. (a) Show that maximum likelihood with the standard model recovers R/T rather than R. (b) Explain why the temperature does not matter for which policy is optimal under the learned reward, but does matter when the learned reward is combined with something else, such as a KL penalty with weight β in RLHF (Section 22.2). (c) Ties: propose an extension of (18.5) that allows a labeler to declare two items equally good.

Exercise 18.13 ● Averaging two good choices

An expert passes an obstacle at lateral position 0 by steering to -1 or to $+1$, each half of the time. (a) What action does behavior cloning with a Gaussian policy of fixed variance (equivalently, the squared loss) learn, and what happens when it is executed? (b) What does a mixture of two Gaussians learn instead? (c) Relate this to the motivation of implicit and diffusion policies in Section 18.8.

Exercise 18.14 ● Programming: preference learning

Extend Task 6 of Lab 18.1. (a) Flip 20% of the preference labels at random and measure how the correlation between the learned and the true reward changes with 300 and 3,000 comparisons. (b) Implement a simple active query strategy: fit two reward models on bootstrap resamples of the data and query the next pair on which they disagree most. Does it reach a correlation of 0.95 with fewer comparisons than random queries? (c) For each learned reward, plan greedily and report the true reward of the plan, as in the lab.

Exercise 18.15 ●●● Research: reward hacking of learned rewards

(a) In the lab, a reward learned from 3,000 comparisons correlates better with the true reward than one learned from 1,000, yet the plan it produces is worse. Explain this in terms of what a planner does with errors in a reward, and design an experiment that measures how the true return of the plan depends on the optimization pressure (for example, by mixing the greedy plan with a reference policy). (b) Summarize the evidence of Gao et al. (2023) on how the true quality of language-model outputs changes as a policy is optimized against a learned reward model, and of Ibarz et al. (2018) on what happens when a policy is trained against a frozen reward model. (c) Propose and justify two mitigations, and say how you would evaluate them.

Part VI

Beyond a Single Agent and a Known State

The Markov decision process assumes one agent that sees the whole state and acts one step at a time. Part VI relaxes each assumption: the agent may see only partial observations and need memory (Chapter 19), its behavior may be organized into skills and goals over many time scales (Chapter 20), and it may share its world with other learning agents (Chapter 21).

CHAPTER 19

Partial Observability and Memory

WHAT YOU WILL LEARN

- Recognize when an agent's observations are not a Markov state, and model such problems as partially observable Markov decision processes (POMDPs).
- Update a belief state by Bayes' rule, and compute beliefs and expected rewards by hand in the Tiger problem.
- Explain why a POMDP is an MDP over beliefs, and what this implies for optimal policies.
- Explain why memoryless policies can fail, why the best of them may be random, and how frame stacking, recurrent networks and transformers give agents memory.
- Prove that finite-horizon value functions over beliefs are piecewise linear and convex, and compute α-vector backups (●●).
- Implement point-based value iteration and POMCP, and state the lower-bound property of point-based methods (●●).
- Discuss information states, predictive state representations and the evidence on recurrent model-free agents as baselines for partially observable problems (●●●).

Every method so far has assumed that the agent sees the state: a description of the present that summarizes everything about the past that matters for the future. Real agents rarely have that luxury. A robot sees the world through a noisy camera and cannot see behind walls. A poker player does not see the other players' cards. A dialogue system does not observe what the user actually wants, only what the user says. A doctor sees symptoms and test results, not the disease. Even in an Atari game, a single frame does not show which way the ball is moving.

When the observation is not a state, the Markov property fails: the best action can depend on what happened before, not just on what the agent sees now. The agent must *remember*, and it must decide what to remember. This chapter develops the two classical answers and their modern descendants. The first is to *infer*: keep a probability distribution over the hidden state, the *belief*, update it by Bayes' rule, and plan over beliefs; this is exact but needs a model and is expensive. The second is to *learn a memory*: give the agent a policy that depends on its history, through a stack of recent observations, a recurrent network or a transformer, and train it end to end; this needs no model but gives few guarantees.

Section 19.1 defines partially observable Markov decision processes (POMDPs) and shows why reacting to the current observation is not enough. Section 19.2 introduces beliefs, works through the classic Tiger problem with numbers, and shows that a POMDP is an MDP over beliefs. Section 19.3 (●●) shows that value functions over beliefs are piecewise linear and convex, which leads to exact value iteration with α-vectors, and Section 19.4 (●●) covers the approximate plan-

ners that scale: point-based value iteration and Monte Carlo tree search over histories (POMCP). Section 19.5 turns to learning without a model: memoryless policies, frame stacking, recurrent policies and transformers. Section 19.6 (●●●) discusses what a good memory should contain. On a first reading, Section 19.1, 19.2 and 19.5 form a complete introduction.

19.1 When the state is hidden

Consider a robot vacuum in a house. Its true state includes its position and the layout of every room, but it observes only a few bumper contacts and a cheap distance sensor. Two different corridors can look identical to it; the same sensor reading can mean "facing a wall" or "facing a sofa". If it acts only on its current reading, it will make the same choice in situations that need different choices. This phenomenon, different states producing the same observation, is called *perceptual aliasing*, and it is the heart of partial observability.

To reason about such problems we keep the MDP of Definition 3.1 for the world, and add a channel through which the agent perceives it.

Definition 19.1 Partially observable Markov decision process

A *partially observable Markov decision process* (POMDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, P, Z, r, \gamma, b_0)$, where $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{O}$ are sets of states, actions and observations; $P(s' \mid s, a)$ is the transition probability; $Z(o \mid s', a)$ is the *observation probability*, the probability of observing o when the environment arrives in state s' after action a; $r(s, a)$ is the expected reward; $\gamma \in [0, 1)$ is the discount factor; and b_0 is the distribution of the initial state. At each step the environment is in a state s_t that the agent does not see; the agent takes a_t, receives r_t, the state moves to $s_{t+1} \sim P(\cdot \mid s_t, a_t)$, and the agent observes $o_{t+1} \sim Z(\cdot \mid s_{t+1}, a_t)$.

An MDP is the special case in which the observation reveals the state ($\mathcal{O} = \mathcal{S}$ and $o_{t+1} = s_{t+1}$). The objective is unchanged: maximize the expected discounted return. What changes is what a policy may depend on. All the agent knows at time t is its *history*

$$h_t = (a_0, o_1, a_1, o_2, \dots, a_{t-1}, o_t),$$

the actions it took and the observations it received (plus its rewards, which in learning problems can carry information too). A policy for a POMDP is therefore, in general, a map from histories to actions, $\pi(a_t \mid h_t)$.

Why reacting is not enough. A *memoryless* or *reactive* policy chooses its action from the current observation alone, $\pi(a \mid o_t)$, exactly as an MDP policy chooses from the state. In an MDP nothing is lost by this. In a POMDP a great deal can be. Two examples make the point. A single Atari frame shows where the ball is but not where it is going; the classic DQN agent (Chapter 10) feeds the network the last four frames for this reason. And in the *aliased corridor* of Section 19.5.1, every cell looks the same, so a memoryless agent that always chooses the same action never reaches the goal. An agent with a little memory solves both: it needs to remember what it has seen and done.

PITFALL

Treating observations as states. Running Q-learning or any MDP algorithm on observations as if they were states is common and sometimes works, but nothing in the theory of the earlier chapters supports it. The value of an observation depends on the policy (which hidden states the agent tends to be in when it sees it), the Bellman equation no longer holds for observations, temporal-difference methods may converge to poor policies or not at all, and the best memoryless policy may be stochastic, which greedy methods cannot represent (Singh et al. 1994).

19.2 Beliefs

If the agent has a model of the POMDP, there is a principled way to remember: keep track of what the history says about the hidden state. The *belief* at time t is the conditional distribution of the state given the history,

$$b_t(s) = \Pr(s_t = s \mid h_t), \qquad s \in \mathcal{S}.$$

It is a vector of $|\mathcal{S}|$ probabilities that sum to one, a point in the *belief simplex* $\Delta(\mathcal{S})$. The belief does not need the whole history to be updated: after taking action a in belief b and observing o, Bayes' rule gives the new belief $b' = \tau(b, a, o)$ as

$$b'(s') = \frac{Z(o \mid s', a) \sum_s P(s' \mid s, a)\, b(s)}{\Pr(o \mid b, a)}, \qquad \Pr(o \mid b, a) = \sum_{s'} Z(o \mid s', a) \sum_s P(s' \mid s, a)\, b(s). \tag{19.1}$$

In words: first *predict*, by pushing the old belief through the dynamics, $\bar{b}(s') = \sum_s P(s' \mid s, a) b(s)$; then *correct*, by multiplying each predicted probability by the likelihood $Z(o \mid s', a)$ of what was actually observed and renormalizing. The denominator is the probability of the observation, which we will need for planning.

The derivation uses only the definition of conditional probability and the Markov structure of the POMDP. Given the history h_t and the action $a_t = a$, the joint probability that the state was s, moves to s' and emits o is $b_t(s)\, P(s' \mid s, a)\, Z(o \mid s', a)$: the state given the history is distributed as b_t, the next state depends only on (s, a), and the observation only on (s', a). Summing over s gives the joint probability of (s', o), and dividing by its sum over s', which is $\Pr(o \mid h_t, a) = \Pr(o \mid b_t, a)$, gives (19.1). This filter is the same one that underlies the Kalman filter in linear-Gaussian systems and particle filters in robotics.

Example 19.1 The Tiger problem

The *Tiger problem* of Kaelbling et al. (1998) is the classic small POMDP. You stand before two closed doors. Behind one is a tiger, behind the other a treasure, with equal probability. You can open the left door, open the right door, or listen. Opening the door with the treasure earns +10, opening the door with the tiger costs 100, and after either the game restarts with the tiger placed behind a random door. Listening costs 1 and does not change anything, but you hear the tiger on the correct side with probability 0.85 and on the wrong side with probability 0.15. We use $\gamma = 0.95$. There are two states (tiger-left, tiger-right), three actions and two observations (hear-left, hear-right), and a belief is a single number, $b = \Pr(\text{tiger-left})$.

Beliefs. Start with $b_0 = 0.5$ and listen. Hearing left has probability $\Pr(\text{hear-left}) = 0.5 \cdot 0.85 + 0.5 \cdot 0.15 = 0.5$, and by (19.1) the new belief is $0.5 \cdot 0.85/0.5 = 0.85$. Listen again and hear left again: this observation had probability $0.85 \cdot 0.85 + 0.15 \cdot 0.15 = 0.745$, and the belief becomes $0.7225/0.745 = 0.9698$. If instead the second observation had been "right", the belief would have returned to $0.85 \cdot 0.15/(0.85 \cdot 0.15 + 0.15 \cdot 0.85) = 0.5$: opposite observations cancel. In general the belief depends only on the difference d between the numbers of "left" and "right" observations since the last door was opened: $b = 0.85^d/(0.85^d + 0.15^d)$.

Expected rewards. In belief b, opening the right door earns $10b - 100(1 - b) = 110b - 100$ in expectation, and opening the left door $10(1 - b) - 100b$. At $b = 0.5$ both doors are worth -45, far worse than listening (-1). At $b = 0.85$ opening the right door is still worth -6.5, but at $b = 0.9698$ it is worth $+6.68$.

The optimal policy. A myopic agent, comparing only immediate rewards, would open the right door as soon as $110b - 100 > -1$, that is, $b > 0.9$; one "left" is not enough, two are. The optimal policy, computed exactly in Section 19.3, is also "open after two net observations on the same side", with thresholds $b > 0.960$ (open right) and $b < 0.040$ (open left); its value is $V^*(0.5) = 19.37$. The optimal threshold is higher than the myopic one because listening once more is cheap and an error is expensive.

♦

Figure 19.1 shows an agent following the optimal policy. Its belief jumps between a few values, 0.5, 0.85, 0.9698 and their mirror images; each pair of agreeing observations ends with a door opened and the belief reset to 0.5.

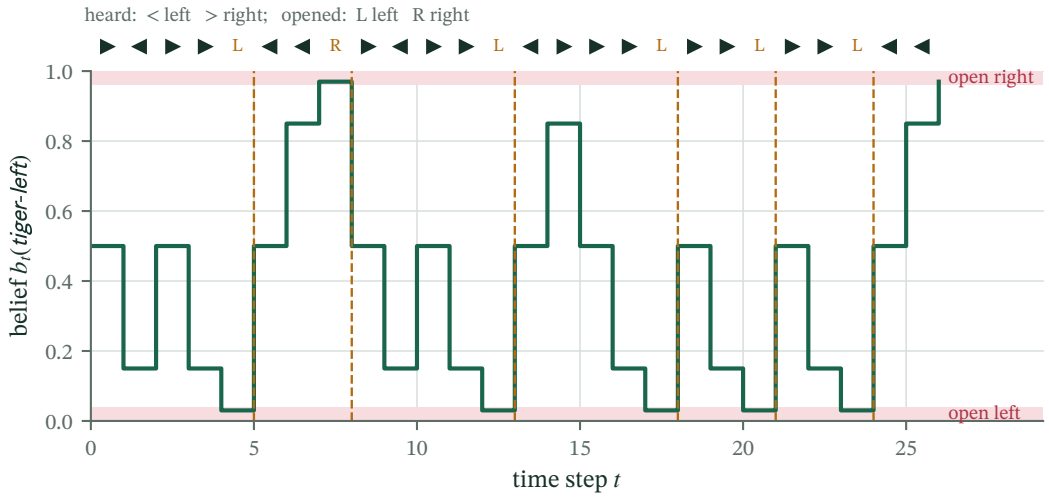


Figure 19.1. The belief of an agent playing the Tiger problem with the optimal policy. Triangles at the top show what it heard; dashed lines mark the doors it opened (all six earned +10 in this episode). The belief moves only when the agent listens, and the agent opens a door only when the belief enters a shaded region ($b > 0.960$ or $b < 0.040$), which takes two net observations on the same side. Conflicting observations cancel and send the belief back to 0.5.

19.2.1 The belief MDP

The belief is more than a convenient summary. It is a *sufficient statistic*: everything the history says about the future is contained in it. Indeed, by (19.1) the probability of the next observation depends on the history only through b_t, the next belief is a function of b_t, the action and the observation, and the expected reward is

$$\mathbb{E}[r_t \mid h_t, a_t = a] = \sum_s b_t(s)\, r(s,a) = r(b_t, a).$$

So the beliefs form a Markov process controlled by the actions. In other words, a POMDP is an MDP whose states are beliefs: the *belief MDP*, with state space $\Delta(\mathcal{S})$, the same actions, reward $r(b,a)$, and transitions that move from b to $\tau(b,a,o)$ with probability $\Pr(o \mid b, a)$, one successor for each observation (Åström 1965; Smallwood and Sondik 1973; Kaelbling et al. 1998).

This reduction brings all the theory of MDPs to bear. The optimal value function over beliefs satisfies the Bellman optimality equation

$$V^*(b) = \max_{a \in \mathcal{A}} \Big[r(b,a) + \gamma \sum_{o \in \mathcal{O}} \Pr(o \mid b, a)\, V^*\big(\tau(b,a,o)\big) \Big].$$

Because rewards are bounded and $\gamma < 1$, the right-hand side defines a γ-contraction on bounded functions of the belief, exactly as in Theorem 4.3, so V^* is its unique solution, and a policy that acts greedily with respect to V^*, a function of the belief alone, is optimal among all history-dependent policies. In the Tiger problem the optimal policy is "listen unless $b > 0.960$ or $b < 0.040$".

The Bellman equation can be checked by hand on the Tiger problem, using values that Section 19.3 computes: $V^*(0.85) = 21.44$ and $V^*(0.5) = 19.37$. At $b = 0.5$ the two observations after listening are equally likely and lead to 0.85 and 0.15, which have the same value by symmetry, so

$$Q^*(0.5, \mathsf{listen}) = -1 + 0.95 \times 21.44 = 19.37, \qquad Q^*(0.5, \mathsf{open}) = -45 + 0.95 \times 19.37 = -26.60,$$

where opening either door earns -45 in expectation and restarts the problem at $b = 0.5$. Listening wins, and its value is $V^*(0.5)$, as the equation requires. At $b = 0.85$ the comparison is closer but still favors listening: $Q^*(0.85, \mathsf{listen}) = 21.44$ against $Q^*(0.85, \mathsf{open\text{-}right}) = -6.5 + 0.95 \times 19.37 = 11.90$.

The catch is that the belief MDP has a *continuous* state space, a simplex of dimension $|\mathcal{S}| - 1$, even when the POMDP is tiny; and a belief over a large state space is itself a large object to compute and store. Exact solution is hopeless beyond small problems: even deciding whether a finite-horizon POMDP has a policy of a given value is PSPACE-complete (Papadimitriou and Tsitsiklis 1987). Section 19.3 and Section 19.4 show how planning over beliefs is done in practice; Section 19.5 shows how to do without beliefs altogether.

CHECKPOINT

1. In the Tiger problem, what is the belief after hearing left, right, left?
2. Why does opening a door reset the belief to 0.5, whatever it was?
3. Is the belief a deterministic function of the history? Is the next belief a deterministic function of the current belief and action?

Answers.

1. 0.85: one "right" cancels one "left", so the net count is one "left".
2. After a door is opened the tiger is placed behind a random door, so the new state is independent of everything before; and the observation after opening carries no information.
3. Yes to the first: b_t is computed from h_t by (19.1). No to the second: the next belief also depends on the observation, which is random; for each observation it is deterministic.

19.3 Value functions over beliefs ●●

What does V^* look like as a function on the belief simplex? The answer is the key to exact POMDP planning: finite-horizon value functions are maxima of finitely many linear functions (Smallwood and Sondik 1973; Sondik 1978).

Plans and α-vectors. Fix a *conditional plan* of depth H: an action to take now, and for each possible observation, a conditional plan of depth $H-1$ to follow next. (Drawn out, it is a tree with actions at the nodes and observations on the edges.) If the agent knew the state s, the expected return of executing the plan p from s would be a number $\alpha_p(s)$; the vector $\alpha_p \in \mathbb{R}^{\mathcal{S}}$ is the plan's *α-vector*. Since the agent does not know the state, the expected return of the plan in belief b is the average

$$V_p(b) = \sum_s b(s)\,\alpha_p(s) = \alpha_p \cdot b,$$

a *linear* function of the belief. The best the agent can do with H steps to go is to choose the best plan for its belief, so $V_H(b) = \max_p \alpha_p \cdot b$ is a maximum of linear functions: piecewise linear and convex. The following theorem makes this precise and shows how to compute the vectors recursively.

Theorem 19.2 Value functions over beliefs are piecewise linear and convex

Let $V_0(b) = \max_{\alpha\in\Gamma_0} \alpha\cdot b$ for a finite set $\Gamma_0 \subset \mathbb{R}^{\mathcal{S}}$ (for example $\Gamma_0 = \{0\}$), and let V_{k+1} be obtained from V_k by the Bellman optimality backup over beliefs. Then for every k there is a finite set Γ_k with $V_k(b) = \max_{\alpha\in\Gamma_k} \alpha \cdot b$. Explicitly,

$$\Gamma_{k+1} = \bigcup_{a\in\mathcal{A}} \Big\{ r_a + \sum_{o\in\mathcal{O}} g_{a,o}^{\alpha_o} \;:\; \alpha_o \in \Gamma_k \text{ for each } o \Big\}, \qquad g_{a,o}^{\alpha}(s) = \gamma \sum_{s'} P(s' \mid s,a)\, Z(o \mid s', a)\, \alpha(s'),$$

where $r_a = r(\cdot, a)$. Consequently V_k is convex for every k, and so is $V^* = \lim_k V_k$.

Proof. By induction on k; the case $k = 0$ is the assumption. Suppose $V_k(b) = \max_{\alpha\in\Gamma_k} \alpha \cdot b$. For an action a and an observation o with $\Pr(o \mid b, a) > 0$, the new belief $\tau(b, a, o)$ is given by

(19.1), and the normalizing constant cancels:

$$\Pr(o \mid b,a)\,V_k(\tau(b,a,o)) = \max_{\alpha\in\Gamma_k} \sum_{s'} \alpha(s')\,Z(o \mid s',a) \sum_{s} P(s' \mid s,a)\,b(s) = \frac{1}{\gamma} \max_{\alpha\in\Gamma_k} g^{\alpha}_{a,o} \cdot b.$$

The same expression is $0 = \Pr(o \mid b,a)V_k(\cdot)$ when $\Pr(o \mid b,a) = 0$, because then every $g^{\alpha}_{a,o} \cdot b$ is zero. So the backup is

$$V_{k+1}(b) = \max_a \Big[r_a \cdot b + \sum_o \max_{\alpha_o\in\Gamma_k} g^{\alpha_o}_{a,o} \cdot b \Big] = \max_a \max_{(\alpha_o)_{o\in\mathcal{O}}} \Big(r_a + \sum_o g^{\alpha_o}_{a,o} \Big) \cdot b,$$

where we exchanged the sum of maxima with the maximum over all choices of one vector per observation, which is allowed because the choices for different o are independent. This is the maximum over the finite set Γ_{k+1}. A maximum of linear functions is convex. Finally, $V_k \to V^*$ uniformly by the contraction property, and a uniform limit of convex functions is convex. ■

Convexity has a nice interpretation: information is valuable. A belief that is a mixture $b = \lambda b_1 + (1-\lambda)b_2$ is less informative than knowing which of b_1 and b_2 holds, and convexity says $V^*(b) \le \lambda V^*(b_1) + (1-\lambda)V^*(b_2)$: on average, the agent does at least as well when it is told which. In the Tiger problem, the corners $b = 0$ and $b = 1$, where the agent knows where the tiger is, are worth the most.

Exact value iteration. The theorem gives an algorithm: start from Γ_0 and repeatedly compute Γ_{k+1}. The set grows quickly, from $|\Gamma_k|$ to $|\mathcal{A}|\,|\Gamma_k|^{|\mathcal{O}|}$ vectors before simplification, but most of the new vectors are *dominated*: they are below the maximum everywhere on the simplex and can be pruned. Pruning in general requires solving a linear program per vector; doing it after each partial cross-sum ("incremental pruning", Cassandra et al. (1997)) keeps the sets small. With two states, as in the Tiger problem, a belief is a single number $x \in [0,1]$, each vector is a line over $[0,1]$, and pruning amounts to computing an upper envelope of lines, which is easy to do exactly. Figure 19.2 shows the result: starting from $\Gamma_0 = \{0\}$, the horizon-1 value is the maximum of the three immediate-reward lines (open left, listen, open right); with more steps to go, new vectors appear in the middle, where listening first pays. Exact value iteration from the lower bound $\min r/(1-\gamma)$ converges for the Tiger problem in 540 iterations to 9 vectors, 7 of which start with listening; at intermediate iterations there were up to 191 vectors.

How hard is it? The number of conditional plans of depth H is $|\mathcal{A}|^{(|\mathcal{O}|^H-1)/(|\mathcal{O}|-1)}$, doubly exponential in the horizon, and in the worst case many of them are needed. This is the *curse of history*, on top of the curse of dimensionality of the belief simplex. Papadimitriou and Tsitsiklis (1987) proved that deciding whether a finite-horizon POMDP has a policy achieving a given value is PSPACE-complete, so a polynomial algorithm is unlikely; and for some criteria without discounting, the corresponding questions for infinite horizons are undecidable (Madani et al. 2003). Exact methods are therefore limited to problems with a handful of states. Even the Tiger problem shows the effect: with a listening accuracy of 0.7 instead of 0.85, exact value iteration already needs about 200 vectors after 30 iterations, and more after each further one (see the first "Going further" item of Lab 19.1).

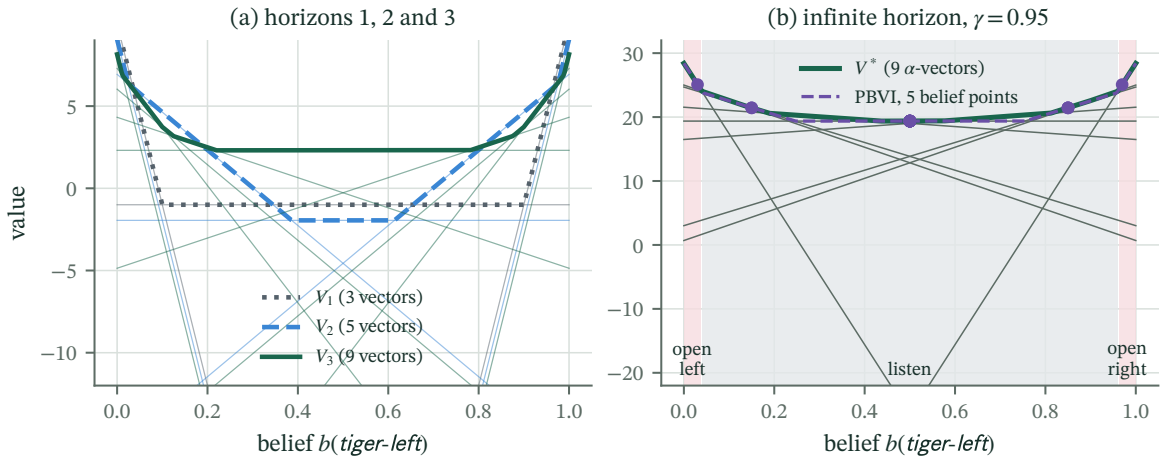


Figure 19.2. Value functions of the Tiger problem over the belief b(tiger-left). (a) Optimal values with 1, 2 and 3 steps to go, starting from zero: each is the upper envelope (thick) of a few α-vectors (thin lines). With one step the agent can only take the best immediate action; with more steps, listening first becomes the best plan near $b = 0.5$. (b) The infinite-horizon value function: the maximum of 9 α-vectors, with the regions where each action is optimal shaded. Point-based value iteration with 5 belief points (dots) is exact at its points and a lower bound everywhere else (dashed).

19.4 Approximate planning ●●

Two ideas made POMDP planning practical for problems with thousands or even billions of states: back up the value function only at beliefs that matter, and plan online, from the current belief only, by simulation.

19.4.1 Point-based value iteration

The exact backup computes vectors that are optimal *somewhere* on the simplex, including regions the agent will never visit. *Point-based value iteration* (PBVI) (Pineau et al. 2003) keeps a finite set B of belief points and computes only one new vector per point: the one that is best *at that point*. For a belief b and the current set Γ, the point-based backup is

$$\text{backup}(b) = \arg\max_{a \in \mathcal{A}} \left(r_a + \sum_o g_{a,o}^{\alpha_{a,o}^b} \right) \cdot b, \qquad \alpha_{a,o}^b = \arg\max_{\alpha \in \Gamma} g_{a,o}^{\alpha} \cdot b,$$

which costs $O(|\mathcal{A}||\mathcal{O}||\Gamma||\mathcal{S}|)$ per point once the projected vectors $g_{a,o}^{\alpha}$ are known (computing them costs $O(|\mathcal{A}||\mathcal{O}||\Gamma||\mathcal{S}|^2)$ per sweep, shared by all points), instead of a number of vectors exponential in $|\mathcal{O}|$.

Algorithm 19.1 Point-based value iteration (PBVI)

Input: a POMDP; an initial set of belief points B (for example $\{b_0\}$); numbers of backups per expansion K and of expansions E.

1. Initialize $\Gamma \leftarrow \{\alpha_0\}$ with $\alpha_0(s) = \min_{s,a} r(s,a)/(1-\gamma)$ for all s (a lower bound).
2. Repeat E times:
   1. Repeat K times: compute $g^{\alpha}_{a,o}$ for all $\alpha \in \Gamma$, a, o; set $\Gamma \leftarrow \{\text{backup}(b) : b \in B\}$.
   2. Expand B: for each $b \in B$, simulate each action once from b (sample a state, a next state and an observation), compute the successor beliefs by (19.1), and add the successor farthest from B (in L^1 distance).

Output: the set Γ; act greedily, \$a = \$ the action of $\arg\max_{\alpha\in\Gamma} \alpha \cdot b$.

Because every vector PBVI computes is the value of an actual conditional plan whose leaves are valued by the lower bound α_0, its value function is a *lower bound* on V^* everywhere (Exercise 19.8), and the bound is tight at the belief points when the points cover the beliefs that matter. The expansion step adds points along trajectories that the agent can actually reach, which is where the approximation needs to be good; Pineau et al. (2003) bound the error by a quantity proportional to $(r_{\max} - r_{\min})\,\delta_B/(1-\gamma)^2$, where δ_B is the largest L^1 distance from a reachable belief to the nearest point of B.

On the Tiger problem the choice of points makes all the difference. The beliefs reached by twenty random steps from b_0 are only five distinct points (0.5, 0.85, 0.9698 and their mirror images), and PBVI on them recovers V^* exactly at those points, $V(0.5) = 19.37$, while staying below V^* elsewhere (by up to 0.92, Figure 19.2b). With only the points $\{0.5, 0.15\}$, PBVI never sees a belief at which opening a door is good, and its best plan is to listen forever, worth $-1/(1-0.95) = -20$. Point-based methods differ mainly in how they choose and order their points: Perseus backs up a random subset of a large fixed set (Spaan and Vlassis 2005), HSVI keeps upper and lower bounds and explores where they disagree (Smith and Simmons 2004), and SARSOP concentrates on beliefs reachable under near-optimal policies (Kurniawati et al. 2008); Shani et al. (2013) compare them.

19.4.2 Online planning with POMCP

Offline planners compute a policy for every belief in advance. An online planner instead thinks only about the current situation, as MPC and Monte Carlo tree search do in fully observable problems (Chapters 8 and 15). *Partially observable Monte Carlo planning* (POMCP) (Silver and Veness 2010) extends Monte Carlo tree search (Algorithm 8.5) to POMDPs with two ideas. The search tree is built over *histories* rather than states: a node is a sequence of actions and observations, and its children are actions, whose children are observations. And the belief at each node is represented by *particles*, states sampled by the simulations that passed through the node, so that no explicit belief update, and not even an explicit model, is needed: only a simulator $(s', o, r) \sim G(s, a)$.

Algorithm 19.2 POMCP

Input: a simulator G; the current history h with particles $B(h)$; number of simulations n; exploration constant c; depth limit D; a rollout policy.

1. Search(h): repeat n times: sample a state s from $B(h)$ and call Simulate(s, h, 0). Return the action a with the largest visit count $N(ha)$.
2. Simulate(s, h, d):
   1. If $d = D$, return 0.
   2. If h is not in the tree: add it, with $N(h) = 0$ and $N(ha) = Q(ha) = 0$ for all a; return Rollout(s, d), the discounted return of the rollout policy from s up to depth D.
   3. Choose $a = \arg\max_a \left[Q(ha) + c\sqrt{\ln N(h)/N(ha)} \right]$ (untried actions first).
   4. Sample $(s', o, r) \sim G(s, a)$; set $G_{\text{sim}} \leftarrow r + \gamma\,\text{Simulate}(s', hao, d+1)$.
   5. Add s to $B(h)$; $N(h) \leftarrow N(h) + 1$; $N(ha) \leftarrow N(ha) + 1$; $Q(ha) \leftarrow Q(ha) + (G_{\text{sim}} - Q(ha))/N(ha)$; return G_{sim}.
3. Act: execute the returned action a, observe o, make hao the new root and keep its particles; if there are too few, add more by *rejection sampling* (simulate a from root particles and keep the next states whose observation equals o).

Output: an action for the current history.

The selection rule in step 2.3 is the UCB1 rule of the bandit chapter (Algorithm 2.2), applied at every node, as in UCT. The particle sets play the role of the belief: a node reached by many simulations has many particles, distributed approximately as the belief at that history, because each simulation passes through a node with a state sampled from the right conditional distribution. POMCP was reported to solve problems with enormous state spaces, such as a partially observable version of Pac-Man, far beyond the reach of the offline methods of the time (Silver and Veness 2010). DESPOT (Somani et al. 2013) searches a sparse tree of sampled "scenarios" and regularizes the policy to avoid overfitting them.

POMCP's weaknesses are those of Monte Carlo search, made sharper by the hidden state. Its value estimates come from rollouts, so a bad rollout policy gives noisy or misleading estimates; and when a belief needs many observations to be pinned down, the particles at deep nodes become few (*particle deprivation*). On the Tiger problem with 2,000 simulations per decision, POMCP listened at $b = 0.5$ in all of 5 runs, whether its rollouts chose random actions or always listened: there, listening is better than opening by more than 40. At $b(\text{tiger-left}) = 0.97$, where opening the right door is optimal but beats listening by less than 1 (out of about 25), it opened the right door in only 2 of 5 runs. Close decisions need many more simulations, or better rollouts.

19.5 Learning without a model

Belief-based planning needs a model of the POMDP, and it needs the model to be small. A learning agent has neither: it does not know P and Z, and the hidden state of a video game or a conversation is not a short list. This section asks what a model-free agent can do. There are two

answers: accept a policy without memory and make the most of it, or give the policy a memory and learn what to put in it.

19.5.1 Memoryless policies

A memoryless policy can still be *stochastic*, and in a POMDP this matters. In an MDP some deterministic policy is always optimal (Theorem 3.7), because the agent always knows where it is. Under aliasing the agent must use the same action distribution in states that need different actions, and randomization can be the best compromise. Singh et al. (1994) showed that the best stochastic memoryless policy can be arbitrarily better than the best deterministic one, and that standard value-based methods applied to observations do not find it.

The *aliased corridor* (Figure 19.3a), Sutton and Barto's "short corridor with switched actions" (Sutton and Barto 2018, Example 13.1), shows how dramatic the difference is. A corridor has three cells and a goal on the right; the agent starts on the left, gets reward -1 per step, and sees the same observation everywhere. In the first and third cells right moves right and left moves left (or bumps into the wall), but in the middle cell the actions are reversed. A deterministic memoryless policy that always goes right shuttles forever between the first two cells; one that always goes left never leaves the first cell. Both earn $-\infty$. A policy that goes right with probability p reaches the goal; solving the three Bellman equations of the induced Markov chain (Exercise 19.10) gives

$$J(p) = -\frac{2(2-p)}{p(1-p)},$$

which is maximized at $p^* = 2-\sqrt{2} \approx 0.586$, with $J(p^*) \approx -11.66$ (Figure 19.3b). An agent with one action of memory does far better: "right first, then the opposite of what you did last" gives right, left, right and reaches the goal in 3 steps, $J = -3$.

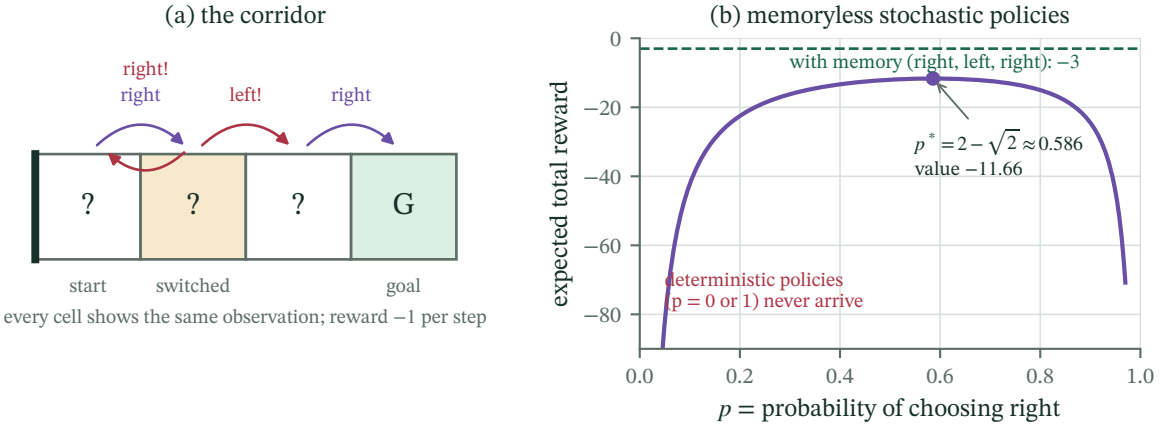


Figure 19.3. The aliased corridor. (a) The agent sees the same observation in every cell, and the middle cell reverses the effect of the actions. (b) The expected total reward of the memoryless policy that chooses right with probability p. Both deterministic policies loop forever; the best memoryless policy is random, $p^* = 2-\sqrt{2}$; and a policy that remembers its last action does much better still (-3, dashed).

The lesson for algorithms is direct. Methods that output greedy policies from action values (Q-learning, SARSA, DQN) cannot represent the best memoryless policy, because they are deter-

ministic; policy-gradient methods (Chapter 11) can, because they optimize stochastic policies directly, and the Monte Carlo (REINFORCE) estimate of the gradient remains unbiased when the policy depends only on observations (a critic defined on observations, however, can be biased). But a stochastic memoryless policy is still a poor substitute for memory.

19.5.2 History-based and recurrent policies

The general solution is to let the policy depend on the history: $\pi(a_t \mid h_t)$. Since histories grow without bound, an agent needs a compact, learnable summary. Three architectures dominate.

Frame stacking: a finite window. The simplest summary is the last k observations, concatenated: $x_t = (o_{t-k+1}, \dots, o_t)$. It turns the POMDP into an MDP whenever the last k observations determine the state. In Atari games four frames suffice to see motion, which is why DQN uses them (Mnih et al. 2015). But a window of k observations cannot remember anything older than k steps, its input grows with k, and the designer must know the memory length in advance.

Recurrent networks: a learned state. A recurrent policy maintains a hidden vector z_t updated by a learned function of the previous hidden vector and the new input, and acts from it:

$$z_t = f_\theta(z_{t-1}, x_t), \qquad \pi_\theta(a_t \mid h_t) = \pi_\theta(a_t \mid z_t),$$

where the input x_t is usually the current observation together with the previous action and reward. In the simplest *Elman network* (Elman 1990), $z_t = \tanh(W_x x_t + W_z z_{t-1} + c)$; gated networks such as the LSTM (Hochreiter and Schmidhuber 1997) add multiplicative gates that make it easier to keep information for a long time. The network is trained by *backpropagation through time* (BPTT): unroll it over a sequence, compute the loss (for example the REINFORCE loss of Algorithm 11.1 or a TD loss), and backpropagate through all the time steps. The gradient that reaches an early input is a product of one Jacobian $\partial z_t / \partial z_{t-1}$ per step, so it can shrink or grow exponentially with the distance; this *vanishing or exploding gradient* problem is what gates were invented to address (Exercise 19.12). The hidden state z_t plays the role of the belief, but nothing forces it to be one: it holds whatever the training signal finds useful.

Hausknecht and Stone (2015) built the *deep recurrent Q-network* (DRQN) by replacing one fully connected layer of DQN with an LSTM, and reported that, seeing a single frame at each step, it performed comparably to DQN with four frames on standard Atari games and degraded more gracefully when frames were randomly blanked out. Recurrent agents need care with replay buffers: stored sequences must start from a sensible hidden state. R2D2 stores the recurrent state with each sequence and "burns in" the network on the first part of the sequence before training on the rest (Kapturowski et al. 2019).

An experiment. The *cue corridor* is a minimal memory task in the spirit of the T-maze of Bakker (2001). At the first step the agent sees a cue, "up" or "down"; then it walks down a corridor of L identical cells, with L drawn between 3 and 8; at the junction it must turn the way the cue said, for a reward of $+1$, or -1 if wrong. Figure 19.4a trains four policies with REINFORCE: a memoryless one, frame stacks of 4 and 10 observations, and an Elman network with 8 hidden units trained by BPTT. Without memory, or with a window of 4 frames, the cue is never visible at the junction and success stays at 50%. A window of 10 frames covers the longest corridor and learns quickly,

slightly faster than the recurrent policy (final success 0.996 against 0.95, averaged over five seeds). The difference appears in Figure 19.4b, which tests the trained policies on corridors of other lengths. The window fails as soon as the corridor is longer than 8, because the cue falls out of the window, and also on corridors shorter than 3, because it places the cue at positions it never saw in training. The recurrent policy learned to store the cue in its hidden state and keep it there, and it succeeds on corridors of length 32, four times longer than any it was trained on.

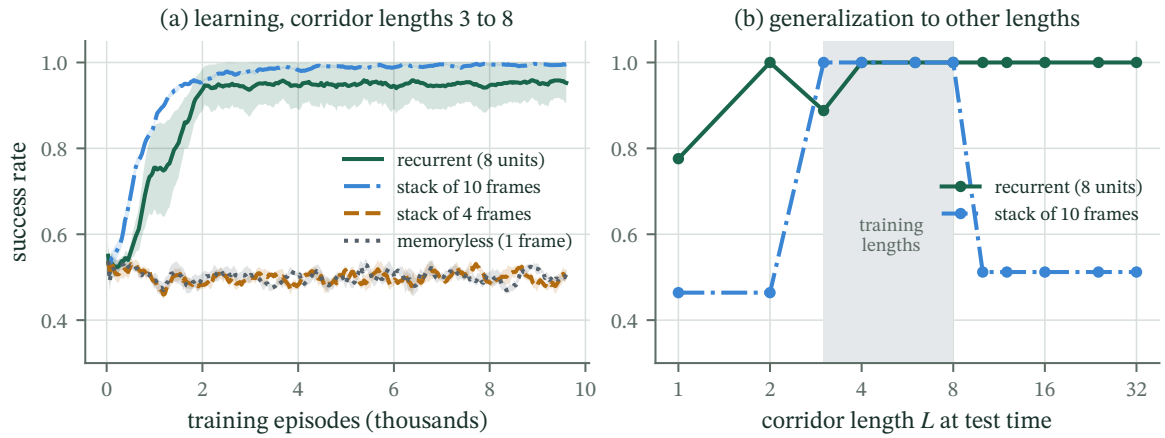


Figure 19.4. Memory on the cue corridor (5 seeds; bands are one standard error). (a) Success rate during training on corridors of length 3 to 8: without memory or with 4 frames, the agent can only guess; with 10 frames or a recurrent network it learns the task. (b) Success of the trained policies on corridors of other lengths: the frame stack fails outside the lengths it was trained on, while the recurrent policy generalizes to much longer corridors.

LAB

Lab 19.1: Beliefs, alpha-vectors and memory (`labs/ch19_pomdp.py`, about 60 minutes; the script runs in about 15 seconds). Run `python3 labs/ch19_pomdp.py`. You will update Tiger beliefs, solve the Tiger problem exactly and with PBVI and POMCP, evaluate memoryless policies in the aliased corridor, and train recurrent and frame-stacking policies on the cue corridor, as in Figures 19.2 to 19.4.

19.5.3 Memory in practice

Most real problems are partially observable to some degree, and a few habits help.

Diagnose first. A sign of missing state is a value function that cannot fit its targets: large, irreducible prediction errors on states that look alike, or returns that vary much more than the value predicts. A quick test is to add information that should not matter in an MDP, such as the previous action, the previous reward or the time step, to the agent's input; if performance improves, the observation was not a state. In particular, when episodes are truncated at a fixed time limit and the remaining time affects what to do, the time step belongs in the observation (Section 6.8).

Try the cheap fixes before the expensive ones. Stacking a few recent observations and feeding the previous action and reward are often enough, and they keep the learner a standard feedforward

agent. Use a recurrent network when the relevant information can be arbitrarily old, or when its age varies, as in the cue corridor.

Recurrent agents have their own pitfalls. Reset the hidden state at the start of every episode, and not in the middle of one. When training from replayed sequences, store the hidden state with the sequence or warm it up on a prefix (the burn-in of R2D2), because a hidden state recomputed from zeros differs from the one the agent had when it acted. Backpropagate through enough steps to span the dependencies that matter: truncating BPTT to k steps silently turns the network into something closer to a k-frame window. And evaluate the agent the way it will be deployed, carrying its hidden state across the steps of an episode.

Belief-based planning is worth it when a model exists. In robotics, dialogue management and many operations problems, a model of the sensors and dynamics is available, the hidden state is small, and the stakes reward reasoning about uncertainty (for example, deciding to gather information before acting, as the Tiger agent does). Point-based and online planners (Section 19.4) then give behavior that a learned policy may take very long to discover.

19.5.4 Transformers as memory ●●

A transformer (Vaswani et al. 2017) offers a third kind of memory. Instead of compressing the past into a fixed vector, it keeps the last K inputs (its *context*) and, at each step, lets the current representation *attend* to all of them, retrieving whatever it needs by content. This removes the recurrent bottleneck and the long chains of Jacobians: information from K steps ago is one attention step away. The price is that memory is again a finite window, as with frame stacking, though with learned, content-based retrieval, and the cost of attention grows quadratically with K.

Transformers were initially hard to train with RL objectives. Parisotto et al. (2020) introduced the gated transformer-XL (GTrXL), which moves the layer normalization and adds gating layers, and reported that it trained stably and outperformed LSTM agents on memory-intensive tasks. The evidence since then is mixed and instructive. Ni et al. (2023) separated two abilities that memory tasks usually conflate: *remembering* information over long spans, at which transformers excel, and *assigning credit* to actions whose consequences come much later, which they reported transformers do not improve. Benchmarks designed specifically for memory, such as POPGym (Morad et al. 2023), have found well-tuned recurrent networks to be strong competitors. Transformers also play a different role in sequence-modeling approaches to RL, where the whole trajectory is the context (Chapter 17).

19.6 Information states and representations ●●●

What *should* a memory contain? The belief answers the question when the model is known, but a learning agent needs a definition that does not mention the hidden state. The classical notion is an *information state*: a statistic $\sigma_t = \sigma(h_t)$ of the history that (i) suffices to predict the expected reward, $\mathbb{E}[r_t \mid h_t, a_t] = \bar{r}(\sigma_t, a_t)$, and (ii) evolves like a state, in the sense that the distribution of σ_{t+1} given (h_t, a_t) depends only on (σ_t, a_t). Any information state supports dynamic programming exactly as beliefs do, and the belief is one, but not the only one: in the

Tiger problem the difference between the numbers of "left" and "right" observations since the last opening is an information state with values in the integers (Exercise 19.13).

Subramanian et al. (2022) turned this into a design principle for learning. An *approximate information state* (AIS) is a learned compression of the history that only approximately satisfies (i) and (ii): it predicts the next reward up to an error ϵ and the distribution of its own next value up to an error δ in a suitable distance between distributions. They proved that dynamic programming on an AIS yields a policy whose suboptimality is bounded by a quantity that grows with ϵ and δ and with the effective horizon $1/(1-\gamma)$, in the spirit of the simulation lemma (Lemma 15.1), and they used the two prediction errors as auxiliary losses for recurrent agents. The view is general: "predict rewards and your own next summary" is also what value-equivalent models (Chapter 15) and many self-predictive representation-learning methods ask for.

Predictive state representations (PSRs) (Littman et al. 2001) go further and define the state as a vector of predictions: the probabilities of a set of *tests*, future action-observation sequences such as "if I listen twice, I hear left twice". The state is then directly verifiable from data, with no hidden variables at all, and Littman et al. (2001) showed that linear PSRs are at least as compact as POMDP models: every POMDP with n states has a linear PSR with at most n core tests. Learning PSRs and related observable-operator models from data, for example by spectral methods, became an active line of work, although they have so far had less impact on deep RL than recurrent agents.

Between explicit beliefs and unconstrained recurrent memories lie hybrid agents. Deep variational RL (Igl et al. 2018) trains a particle filter inside the agent, with a learned generative model, as an approximate belief. World models such as Dreamer (Section 15.6) maintain a latent state that is trained to predict observations and rewards, which is a learned approximate information state. A useful sanity check on all such work came from Ni et al. (2022), who reported that a carefully implemented recurrent model-free agent (with design choices such as separate recurrent encoders for the actor and the critic and suitable context lengths) matched or exceeded specialized methods on many partially observable benchmarks, so that new methods for POMDPs should be compared against strong recurrent baselines.

RESEARCH NOTE

Open questions: What is the right inductive bias for memory in RL: recurrent states, attention over long contexts, explicit beliefs, or predictive objectives? How should memory and long-term credit assignment, two different abilities, be learned together? How can an agent know which past events will turn out to matter? With language-model agents, whose "memory" is a text context, these questions have returned at scale; Ni et al. (2023), Morad et al. (2023) and Subramanian et al. (2022) are good starting points.

HISTORICAL NOTE

Åström (1965) introduced the belief as the state of a partially observed control problem. Smallwood and Sondik (1973) showed that finite-horizon value functions are piecewise linear and convex and gave the first exact algorithm, and Sondik (1978) treated the infinite horizon. Kaelbling et al. (1998) brought POMDPs to artificial intelligence, with the Tiger problem as their running example; point-based methods (Pineau et al. 2003) and online Monte Carlo planning (Silver and Veness 2010) made them practical for larger problems.

19.7 Summary

- When observations do not reveal the state, the problem is a partially observable MDP: an MDP for the world plus an observation channel $Z(o \mid s', a)$. Policies must in general depend on the whole history.
- Memoryless policies can fail badly under perceptual aliasing; in the aliased corridor both deterministic memoryless policies never reach the goal, the best memoryless policy is random ($p^* = 2 - \sqrt{2}$, value -11.66), and one step of memory gives -3.
- The belief, the conditional distribution of the hidden state, is updated by Bayes' rule (19.1): predict with the dynamics, correct with the observation likelihood, normalize.
- In the Tiger problem, one observation moves the belief to 0.85 and two agreeing observations to 0.9698; the optimal policy listens until two net observations agree and then opens the other door, with $V^*(0.5) = 19.37$.
- A POMDP is an MDP over beliefs; its optimal value satisfies a Bellman equation over the belief simplex, and a policy greedy with respect to it is optimal.
- (●●) Finite-horizon value functions are piecewise linear and convex: maxima of α-vectors, the values of conditional plans. Exact value iteration computes them but suffers from the curses of dimensionality and history; finite-horizon planning is PSPACE-complete.
- (●●) Point-based value iteration backs up only at selected beliefs and gives lower bounds; POMCP plans online by Monte Carlo tree search over histories with particle beliefs.
- Model-free agents get memory from frame stacks (a fixed window), recurrent networks trained by backpropagation through time (a learned state), or transformers (attention over a window). In the cue corridor, a recurrent policy generalized to corridors four times longer than in training, while a frame stack failed as soon as the cue left its window.
- (●●●) Information states formalize what a memory must contain; approximate information states, predictive state representations and strong recurrent baselines frame current research.

The next chapter turns from structure in what the agent sees to structure in what it does: temporally extended actions, goals and hierarchies.

19.8 Further Reading

Kaelbling et al. (1998) remains the best introduction to POMDPs: it explains beliefs, the belief MDP, policy trees and α-vectors with the Tiger problem, and it surveys exact algorithms. For the original theory, read Åström (1965) on beliefs, Smallwood and Sondik (1973) on the

piecewise-linear convex structure, and Sondik (1978) on the infinite horizon. Cassandra et al. (1997) describe incremental pruning, the standard exact algorithm, and Papadimitriou and Tsitsiklis (1987) prove the complexity results.

For approximate planning, start with PBVI (Pineau et al. 2003) and then the survey of point-based solvers by Shani et al. (2013), which covers Perseus, HSVI and SARSOP. Silver and Veness (2010) is a short and clear paper on POMCP, and Somani et al. (2013) present DESPOT, its main successor for online planning.

For learning, Singh et al. (1994) is the classic analysis of memoryless policies, and Chapter 17 of Sutton and Barto (2018) discusses partial observability and approximate state from the learning point of view. Hausknecht and Stone (2015) (DRQN) and Kapturowski et al. (2019) (R2D2) show how recurrent networks are combined with deep Q-learning, and Parisotto et al. (2020) how transformers can be. Ni et al. (2022) is required reading before proposing a new method for POMDPs, and Ni et al. (2023) and Morad et al. (2023) are good guides to evaluating memory.

For the representation question, Subramanian et al. (2022) develop approximate information states rigorously and connect them to learning; Littman et al. (2001) introduced predictive state representations; and Igl et al. (2018) train approximate beliefs inside deep agents.

19.9 Exercises

Exercise 19.1 • Tiger beliefs by hand

Starting from $b(\text{tiger-left}) = 0.5$, the agent listens four times and hears left, left, right, left. (a) Compute the belief and the probability of each observation, step by step. (b) Repeat with a listening accuracy of 0.7 instead of 0.85. (c) Show that the belief after any sequence of listening observations depends only on the difference d between the numbers of "left" and "right", and give the formula.

Exercise 19.2 • Why listen more than necessary?

(a) Show that a myopic agent, maximizing only the immediate expected reward, opens the right door in the Tiger problem when $b(\text{tiger-left}) > 0.9$. (b) The optimal policy opens it only when $b > 0.960$. With one "left" observation the belief is 0.85; with two it is 0.9698. Which beliefs does the agent actually reach, and do the two thresholds lead to different behavior? (c) Explain in words why the optimal threshold is higher than the myopic one.

Exercise 19.3 •• The belief MDP

Prove the facts used in Section 19.2.1: (a) $\Pr(o_{t+1} = o \mid h_t, a_t = a) = \Pr(o \mid b_t, a)$ as defined in (19.1); (b) $\mathbb{E}[r_t \mid h_t, a_t = a] = \sum_s b_t(s)\, r(s,a)$; (c) the belief b_{t+1} is a deterministic function of (b_t, a_t, o_{t+1}). Conclude that the process (b_t) with actions (a_t) is a Markov decision process, and write its transition probabilities.

Exercise 19.4 ●● Convexity without alpha-vectors

Give a direct proof that V^* is convex on the belief simplex, without using Theorem 19.2: (a) show that for a fixed history-dependent policy π, the expected return from initial belief b is $\sum_s b(s)V^\pi(s)$, where $V^\pi(s)$ is the expected return of π when the initial state is s (the policy cannot see the state); (b) deduce that $V^*(b) = \sup_\pi \sum_s b(s)V^\pi(s)$ is convex. (c) Interpret the inequality $V^*(\lambda b_1 + (1-\lambda)b_2) \le \lambda V^*(b_1) + (1-\lambda)V^*(b_2)$ as a statement about the value of information.

Exercise 19.5 ●● An alpha-vector backup

In the Tiger problem with $\Gamma_1 = \{(10, -100), (-1, -1), (-100, 10)\}$ (the immediate rewards of open right, listen and open left, as vectors over (tiger-left, tiger-right)), compute the horizon-2 α-vector of the plan "listen; then open the right door if you heard left, and listen if you heard right", using the formula of Theorem 19.2. At which beliefs is this plan the best two-step plan? Check your answer against Figure 19.2a.

Exercise 19.6 ●● Three steps of Tiger

Starting from $b = 0.5$ with three decisions to make (and no value afterward), show that the best plan is to listen twice and then open a door only if both observations agreed, and that its expected return is $V_3(0.5) = 2.3098$. Why is $V_2(0.5) = -1.95$ negative while $V_3(0.5)$ is positive?

Exercise 19.7 ● Choosing belief points

Explain why PBVI on the belief points $\{0.5, 0.15\}$ finds the value -20 at $b = 0.5$, the value of listening forever, although opening after two agreeing observations is worth 19.37. Which belief points must be included for PBVI to find the optimal policy, and how would PBVI's expansion step find them?

Exercise 19.8 ●● PBVI gives lower bounds

Prove that if every vector in Γ satisfies $\alpha \cdot b \le V^*(b)$ for all beliefs b, then so does every vector produced by the point-based backup, and conclude that PBVI's value function is a lower bound on V^* at every iteration. (Hint: show that each new vector is the value of a plan whose leaves are valued by vectors of Γ, or use the monotonicity of the Bellman backup.)

Exercise 19.9 ●● Particles and rejection sampling

In POMCP, the particles for the new root are obtained by simulating the chosen action from old particles and keeping the next states whose observation matches the real one. (a) In the Tiger problem with belief $b(\text{tiger-left}) = 0.85$, what fraction of simulated particles is accepted when the agent listens and hears right? What is the distribution of the accepted particles? (b) What happens after three agreeing observations ($b = 0.9945$) if the fourth observation disagrees? (c) Describe *particle deprivation* and one remedy.

Exercise 19.10 ●● The aliased corridor

(a) For the memoryless policy that chooses right with probability p, write the Bellman equations for the expected total rewards V_1, V_2, V_3 of the three cells and solve them to obtain $J(p) = V_1 = -2(2-p)/(p(1-p))$. (b) Show that J is maximized at $p^* = 2 - \sqrt{2}$ and compute $J(p^*)$. (c) Show that the ε-greedy policies with respect to "always right" ($p = 1 - \varepsilon/2$) are much worse than p^* for small ε. (d) Give a policy with one bit of memory that reaches the goal in 3 steps.

Exercise 19.11 ● What a window can and cannot do

(a) Why are four frames enough for a DQN agent to perceive the velocity of the ball in an Atari game? (b) How many frames are needed to solve the cue corridor with lengths between 3 and 8, and why did the 10-frame policy of Figure 19.4 fail on corridors of length 1 and 2? (c) Give an example of a task in which no fixed window suffices.

Exercise 19.12 ●● Backpropagation through time

For the Elman policy $z_t = \tanh(W_x x_t + W_z z_{t-1} + c)$ with logits $W_o^{\mathsf{T}} z_T$ at the last step, (a) derive the gradient of a loss ℓ of the logits with respect to W_z as a sum over time of contributions, and check it against the method `ElmanPolicy.grads` of the lab code. (b) Show that $\|\partial z_T / \partial z_0\| \le \|W_z\|^T$ (spectral norm) and explain what this implies for learning to remember the cue when $\|W_z\| < 1$.

Exercise 19.13 ●● An information state for the Tiger

Show that in the Tiger problem the integer d_t, the number of "left" minus the number of "right" observations since the last door was opened, is an information state: the expected reward and the distribution of d_{t+1} depend on the history only through (d_t, a_t). Write the transition probabilities of d_t under listening. Why is d_t a more convenient state than the belief for a tabular learning agent, and what would go wrong if the agent counted observations since the start of the episode instead?

Exercise 19.14 ●●● The curse of history

(a) Show that the number of distinct conditional plans of depth H is $|\mathcal{A}|^{(|\mathcal{O}|^H - 1)/(|\mathcal{O}|-1)}$, and evaluate it for the Tiger problem with $H = 3$ and $H = 5$. (b) In the Tiger problem exact value iteration needs only 9 vectors at $H = 3$. Why can pruning reduce the count so much here, and why does it help less when the listening accuracy is lower? (c) Read the statement of the PSPACE-completeness result of Papadimitriou and Tsitsiklis (1987) and explain in a paragraph what it implies, and does not imply, for practical POMDP planning.

Exercise 19.15 ● Programming: how good must your hearing be?

Using `pm.tiger(accuracy=...)` and `pm.pbvi` on a grid of 101 belief points (exact value iteration becomes slow when the accuracy is low), find the optimal opening threshold, the number of net agreeing observations needed before opening, and $V^*(0.5)$ for accuracies 0.95, 0.85 and 0.7. Explain the pattern.

Exercise 19.16 ●● Programming: what makes a memory learnable?

Train the Elman policy of Lab 19.1 on the cue corridor (300 updates, four seeds) with 8 hidden units, with 2 hidden units, and with 8 units but the initial recurrent matrix scaled by 0.5 and by 0.2 (`recurrent_scale`). Report the success rates and explain the differences using Exercise 19.12.

Exercise 19.17 ●●● Memory in large-scale RL

Write a short research memo on how large-scale RL agents should remember. Compare recurrent agents (DRQN, R2D2), transformer agents (GTrXL) and agents with explicit or learned beliefs (DVRL, world models), using Ni et al. (2022), Ni et al. (2023) and Morad et al. (2023). Distinguish memory from credit assignment, propose an experiment that would separate them for a method of your choice, and discuss what changes for language-model agents whose memory is a text context.

CHAPTER 20

Hierarchy, Goals, and Temporal Abstraction

WHAT YOU WILL LEARN

- Explain how acting and planning at several time scales can help with long horizons, exploration and transfer, and what it costs.
- Define an option, compute its model, and use the semi-Markov Bellman equation to plan with options.
- Implement SMDP Q-learning and intra-option Q-learning, and handle a time limit that cuts an option off.
- Describe the main ways to obtain options and skills: by design, from bottlenecks, by gradient ascent (the option-critic) and from mutual information (●●).
- Implement goal-conditioned Q-learning with hindsight experience replay, and state when relabeling is correct.
- Use successor representations and successor features to evaluate and transfer policies, and prove the generalized policy improvement theorem (●●).
- Assess what hierarchical methods have achieved and where their open problems lie (●●●).

When you travel to a conference, you do not plan in muscle twitches. You plan "book a flight, take a taxi to the airport, find the gate", and each of these steps unfolds into many smaller ones that you rarely think about in advance. The agents of the previous chapters had no such luxury. At every time step they chose one primitive action, and they learned values of one-step decisions. When a task takes thousands of steps, learning at that resolution is slow: reward information creeps backward one step per update, and random exploration rarely gets far from where it started.

This chapter is about structure over time. Its central object is the *option* (Sutton et al. 1999): a policy together with a rule for when to stop. An agent that chooses among options makes a few large decisions instead of many small ones. Formally, options turn a Markov decision process into a *semi-Markov decision process*, in which a decision can take a variable amount of time, and almost everything from Chapters 3 to 6 carries over with one change: the discount applied to a decision depends on how long it took. Two further kinds of structure complete the picture. *Goals*: a single goal-conditioned value function can serve every goal at once, and *hindsight* lets an agent learn from its failures by pretending that it meant to reach the place where it actually ended up. And *predictive representations*: the *successor representation* separates what a policy does to the state from what the rewards are, so that the values of new tasks come almost for free.

Hierarchical reinforcement learning has a long history and a mixed record. Some of its ideas, such as goal-conditioned policies and hindsight relabeling, are now standard tools. Others, above all the automatic discovery of useful options, remain open problems. We present the ideas that are solid, show them working on small problems where every number can be checked, and say plainly where the evidence is thin.

Section 20.1 motivates temporal abstraction with the four-rooms world that runs through the chapter. Section 20.2 defines options and semi-Markov decision processes and develops planning and learning with options; its last subsection, on intra-option learning, is ●●. Section 20.3 asks where options come from; its subsections on the option-critic architecture, bottlenecks and skill discovery are ●●. Section 20.4 describes manager–worker hierarchies, and Section 20.5 covers goal-conditioned reinforcement learning and hindsight experience replay. Section 20.6 (●●) develops the successor representation and transfer with successor features, and Section 20.7 (●●●) weighs the evidence for hierarchy, including its return in agents built on language models. A first reading can stop after Section 20.5, skipping the ●● subsections.

20.1 Why structure over time helps

Think about how you give directions inside a building: "leave this room by the door on the right, follow the corridor, take the second door". Nobody says "step, step, step, turn". Good directions name a handful of *temporally extended* behaviors, each of which takes many steps and ends in a recognizable situation. People and animals organize behavior this way at every scale, from reaching and grasping to cooking a meal, and the same organization helps artificial agents. A household robot that has learned to open doors, pick up cups and walk to the kitchen can be asked to make tea by combining those skills; it does not have to rediscover walking for every new errand.

The running example of this chapter is the *four-rooms world* of Sutton et al. (1999), shown in Figure 20.1. An agent moves on a grid of 104 cells divided into four rooms, which are joined by four one-cell hallways. In every cell it can try to move up, right, down or left; a move into a wall leaves it where it is. Reaching the goal cell G in the bottom-right room ends the episode with reward $+1$; every other move earns 0. The discount is $\gamma = 0.9$, so the optimal value of a cell is γ^{d-1} when the goal is d moves away. The agent starts in the top-left corner S. In the *deterministic* version moves always succeed. In the *slippery* version a move goes to one of the two perpendicular directions with probability $1/3$ ($1/6$ each). The original paper used a slightly different noise model, in which the agent slips in any of the other three directions.

The shortest route from S to G takes 16 moves, so the optimal value of the start is $V^*(\mathrm{S}) = 0.9^{15} \approx 0.206$. Described with larger units, the same route is four decisions: "go to the hallway on the right", "go to the hallway below", "down", "down". This example already shows the four benefits that temporal abstraction can bring.

Shorter horizons for credit assignment. Value information moves backward one decision per update. With primitive actions, the reward at G must be passed back through 16 decisions before the start learns about it; with hallway options it is passed back through four, a whole room at a

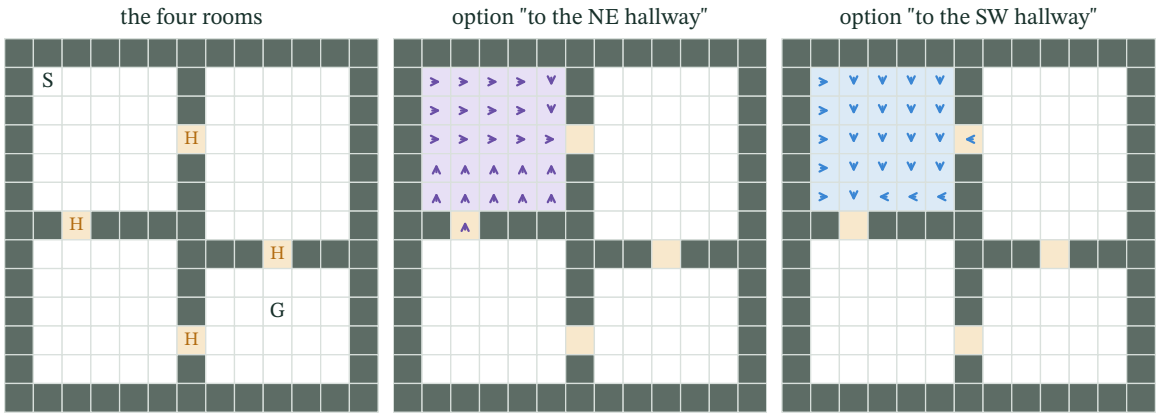


Figure 20.1. The four-rooms world (left) and two of its eight *hallway options* (middle and right). Each option can be started anywhere in its room (shaded) or in the room's other hallway, follows the arrows toward its target hallway, and stops as soon as the agent leaves the room. H marks the hallways, S the start and G the goal.

time. Section 20.2.3 makes this precise: value iteration with primitive actions needs 16 sweeps before the value of S becomes positive, and with hallway options it needs 4.

Exploration at a larger scale. A random choice among skills can carry the agent far; a random choice among primitive actions makes it jitter in place. A uniformly random policy in the deterministic world needs on average about 806 moves to stumble from S into G. Learning agents that explore by choosing options at random find the goal much sooner: in the experiment of Section 20.2.4, their first episodes are about five times shorter.

Reuse and transfer. "Go to the hallway on the right" is useful whatever the goal is. When the task changes, skills that were learned for the old task can be kept, and only the decision of which skill to use when must be relearned.

Planning with fewer, larger steps. If the agent knows what each skill does (where it tends to end, how long it takes and how much reward it collects on the way), it can plan with the skills as if they were actions, looking ahead a few large steps instead of many small ones.

There are costs as well. If the agent may use *only* the options, it may be unable to behave optimally: with the hallway options alone, the deterministic agent can never reach a goal in the middle of a room. Adding options to the primitive actions removes this danger but gives the agent more choices at every decision, which can slow learning when the options are useless. And somebody has to design the options, or the agent must discover them, which is the hardest part.

KEY IDEA

Temporal abstraction replaces many small decisions by a few large ones. It helps when the large decisions are good building blocks for the task, it costs little when the primitive actions remain available, and finding good building blocks is the hard part.

20.2 Options and semi-Markov decision processes

20.2.1 Options

An option packages three things: where a behavior may start, what it does, and when it ends.

Definition 20.1 Option

An *option* o in an MDP with states $\mathcal{S}$ and actions $\mathcal{A}$ is a triple $(\mathcal{I}_o, \pi_o, \beta_o)$: an *initiation set* $\mathcal{I}_o \subseteq \mathcal{S}$ of states in which the option may be started, an *intra-option policy* $\pi_o(a \mid s)$ that chooses primitive actions while the option runs, and a *termination function* $\beta_o : \mathcal{S} \to [0,1]$. An option started in $s_t \in \mathcal{I}_o$ takes $a_t \sim \pi_o(\cdot \mid s_t)$; in every state s_{t+1} that it reaches it stops with probability $\beta_o(s_{t+1})$, and otherwise it takes $a_{t+1} \sim \pi_o(\cdot \mid s_{t+1})$, and so on.

Note the timing: the termination function is consulted in each state the option *reaches*, after its first action, never in the state where it starts. An option therefore lasts at least one step. Its *duration*, the number of steps $k \geq 1$ until it stops, is in general random.

Three examples show the range of the definition.

- Primitive actions are options. The action a is the option with $\mathcal{I} = \mathcal{S}$, $\pi(a \mid s) = 1$ in every state, and $\beta \equiv 1$: it takes a once and stops. A set of options can therefore always include the primitive actions.
- Hallway options. In the four-rooms world, each room has two hallways, and for each of them there is an option that leads to it (Figure 20.1). Its initiation set is the room plus the room's other hallway; its policy moves along a shortest path to the target hallway; and its termination function is 0 inside the room and 1 everywhere else, so it stops as soon as the agent leaves the room, normally in the target hallway. There are eight such options, two per room.
- An option on the Trail. The option "go to Goal" can be started in Start or Middle, always chooses go, and stops in Goal: $\beta(\text{Start}) = \beta(\text{Middle}) = 0$ and $\beta(\text{Goal}) = 1$.

An option whose policy and termination depend only on the current state is called *Markov*. Some useful options are not: "walk for ten seconds" must count time, and "try to open the door, and give up after three attempts" must remember its attempts. Such *semi-Markov* options may depend on the whole history since the option started. Most of the theory below holds for both kinds; we say when it does not.

An agent that uses options chooses among them with a *policy over options*, $\mu(o \mid s)$, defined over the options whose initiation set contains s. (We write μ for it, to distinguish it from the intra-option policies π_o.) In the simplest execution model, called *call-and-return*, the agent picks an option, follows it until it stops, and then picks again. Seen at the level of primitive actions, this behavior is not Markov in s alone, because the action taken in a state depends on which option is running; it is Markov in the pair (s, o).

20.2.2 Semi-Markov decision processes

From the point of view of the policy over options, decisions happen at irregular moments, and between decisions the world runs on its own (Figure 20.2). A decision model in which each decision takes a random amount of time is a *semi-Markov decision process* (SMDP); SMDPs were studied in operations research long before options (see Puterman 1994, ch. 11). The basic fact, proved by Sutton et al. (1999), is that an MDP together with a set of options, used in call-and-return fashion, is an SMDP. Decisions are made only when an option stops; each decision leads, after a random duration, to a random next decision state, and earns the discounted reward collected along the way.

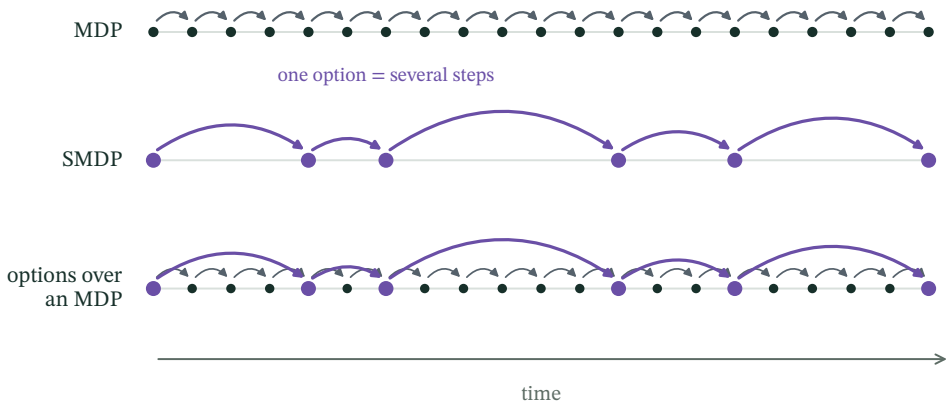


Figure 20.2. Three views of time. In an MDP (top) a decision is made at every step. In a semi-Markov decision process (middle) each decision lasts a variable number of steps. Options over an MDP (bottom) are both at once: the agent decides among options at the large dots, while the primitive steps underneath remain visible and can be learned from.

Everything we need to know about an option, for planning with it, is summarized by its *model*. Suppose option o is started in state s at time t and stops after k steps. Its *reward model* is the expected discounted reward collected while it runs, and its *transition model* is the discounted distribution of where it stops:

$$R_o(s) = \mathbb{E}\big[r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} \,\big|\, s_t = s\big], \qquad P_o^\gamma(s' \mid s) = \sum_{j=1}^{\infty} \gamma^j \Pr\big(k = j,\ s_{t+j} = s' \,\big|\, s_t = s\big). \tag{20.1}$$

Both expectations are over the randomness of the environment, of the intra-option policy and of the termination. The transition model folds the discounting into the probabilities: $P_o^\gamma(s' \mid s)$ is not a probability distribution but a *discounted* one, whose total mass $\sum_{s'} P_o^\gamma(s' \mid s) = \mathbb{E}[\gamma^k]$ is at most γ, and the smaller the longer the option tends to run. This is the one real change from an MDP: a decision that takes k steps discounts what follows by γ^k, not by γ. Sutton et al. (1999) call (R_o, P_o^γ) a *multi-time model*. For a primitive action a, which always lasts one step, it reduces to the familiar $R_a(s) = r(s,a)$ and $P_a^\gamma(s' \mid s) = \gamma P(s' \mid s, a)$.

Example 20.1 The model of an option on the slippery Trail

On the slippery Trail each go fails with probability 0.2 and leaves the hiker where she is. Take the option "go to Goal" and $\gamma = 0.9$. While it runs no reward is collected (all go moves earn 0), so $R_o(\text{Start}) = R_o(\text{Middle}) = 0$.

From Middle, the number of attempts until a go succeeds is geometric: it is j with probability $0.2^{j-1} \cdot 0.8$. The option stops in Goal after j steps, so

$$P_o^\gamma(\text{Goal} \mid \text{Middle}) = \sum_{j=1}^{\infty} \gamma^j \, 0.2^{j-1} \, 0.8 = \frac{0.8\,\gamma}{1 - 0.2\,\gamma} = \frac{0.72}{0.82} = \frac{36}{41} \approx 0.878.$$

From Start the hiker must first reach Middle, which takes an independent geometric number of steps with the same distribution, so $P_o^\gamma(\text{Goal} \mid \text{Start}) = (36/41)^2 \approx 0.771$. The value of starting the option in Start is then $0.771 \times V(\text{Goal}) = 0.771 \times 4 \approx 3.084$, which is exactly $V^*(\text{Start})$ for the slippery Trail. On the ordinary Trail the same option has $P_o^\gamma(\text{Goal} \mid \text{Start}) = \gamma^2 = 0.81$ and value 3.24.

♦

With models in hand, the Bellman equations of Chapter 3 carry over to options. Let $\mathcal{O}$ be a set of options and $\mathcal{O}_s$ those available in s. The optimal value over options, $V_{\mathcal{O}}^*(s)$, is the largest expected return that a policy over $\mathcal{O}$ can achieve from s. It satisfies the *SMDP Bellman optimality equation*

$$V_{\mathcal{O}}^*(s) = \max_{o \in \mathcal{O}_s} \Big[R_o(s) + \sum_{s'} P_o^\gamma(s' \mid s)\, V_{\mathcal{O}}^*(s') \Big], \tag{20.2}$$

and the option values $Q_{\mathcal{O}}^*(s, o)$ satisfy

$$Q_{\mathcal{O}}^*(s, o) = R_o(s) + \sum_{s'} P_o^\gamma(s' \mid s) \max_{o' \in \mathcal{O}_{s'}} Q_{\mathcal{O}}^*(s', o'). \tag{20.3}$$

In words: the value of starting an option is the reward it collects while it runs plus the value of the state where it stops, discounted by γ raised to the option's duration. Compare (3.10): the equation has the same shape, with the one-step reward and the one-step transition replaced by the option's multi-time model. Exercise 20.4 derives it from the definition of the return.

Two facts tell us what options can and cannot do to the optimal value.

Proposition 20.2 Options never beat the optimal policy, and with the primitives they match it

For every set of options, $V_{\mathcal{O}}^*(s) \le V^*(s)$ in every state. If $\mathcal{O}$ contains every primitive action as a one-step option, then $V_{\mathcal{O}}^* = V^*$.

Proof. A policy over options, executed step by step, is a policy in the original MDP: it chooses each primitive action from the history so far (which includes the running option and its start). By Theorem 3.7, no policy, history-dependent or not, earns more than V^*; this gives the inequality. If the primitive actions are among the options, the optimal deterministic policy of the MDP is itself a policy over (one-step) options, so $V_{\mathcal{O}}^* \ge V^*$. ■

The SMDP Bellman operator, which maps V to the right-hand side of (20.2), is a γ-contraction in the maximum norm, just like the operator $\mathcal{T}^*$ of Chapter 4: every option lasts at least one step, so $\sum_{s'} P_o^\gamma(s' \mid s) \le \gamma$ and the argument of Theorem 4.3 goes through unchanged. Value iteration and policy iteration therefore work with options exactly as they work with actions.

20.2.3 Planning with options

SMDP value iteration is value iteration (Algorithm 4.3) with the backup (20.2). For each option we first compute its model, which for Markov options amounts to solving two linear systems (the lab does this, and Exercise 20.5 derives the equations). Then we sweep over the states, replacing $V(s)$ by the right-hand side of (20.2) until nothing changes.

Figure 20.3 runs SMDP value iteration on the deterministic four-rooms world, once with the four primitive actions only and once with the eight hallway options added. Both converge to the same V^*, as Proposition 20.2 promises; with options, value iteration even stops after 7 sweeps instead of 17. The difference is in how fast the value spreads. With primitive actions, each sweep extends the region with positive value by one cell, and the start cell learns about the goal only in sweep 16. With options, once the hallway next to the goal has a value, a single backup carries it across the whole adjacent room: the start cell has its final value $0.9^{15} \approx 0.206$ after 4 sweeps.

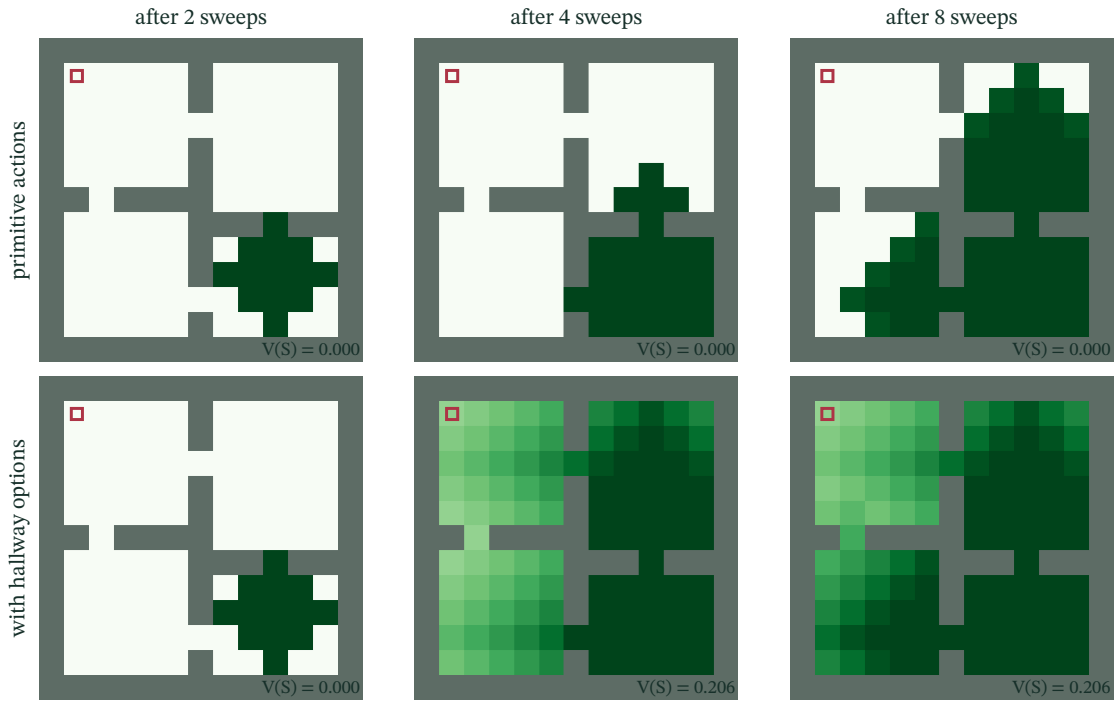


Figure 20.3. Value iteration in the deterministic four-rooms world after 2, 4 and 8 sweeps, with primitive actions only (top) and with the hallway options added (bottom). Darker green is higher value; the small square marks the start S. With options, value spreads a room at a time, and the start has its optimal value 0.206 after four sweeps; with primitive actions alone it is still zero after eight.

Where the goal is matters. If the goal is the hallway between the two right-hand rooms, the hallway options alone are enough: planning with them and no primitive actions gives exactly

V^* in the deterministic world, after 3 sweeps instead of the 15 that primitive actions need. If the goal is inside a room, as in our default, the hallway options alone can never reach it, and the planner needs the primitive actions for the last few steps. This is the general pattern: options give fast, coarse planning, and primitive actions give the final precision.

20.2.4 SMDP Q-learning

Without a model, an agent can learn option values from experience exactly as Q-learning learns action values (Algorithm 6.3). The only differences are that an "action" now runs for several steps and that the target discounts by γ^k. The method goes back to Bradtke and Duff (1995), who studied continuous-time SMDPs; Sutton et al. (1999) used it with options.

Algorithm 20.1 SMDP Q-learning

Input: a set of options $\mathcal{O}$, step size $\eta \in (0, 1]$, exploration rate ε, discount γ.

1. Initialize $Q(s, o)$ for all s and $o \in \mathcal{O}_s$, for example to 0.
2. For each episode:
    1. Observe the initial state s.
    2. Repeat until the episode ends:
        1. Choose $o \in \mathcal{O}_s$ ε-greedily with respect to $Q(s, \cdot)$.
        2. Run o until it stops, in s' after k steps; let $G = r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1}$.
        3. $y \leftarrow G$ if s' is terminal, otherwise $y \leftarrow G + \gamma^k \max_{o' \in \mathcal{O}_{s'}} Q(s', o')$.
        4. $Q(s, o) \leftarrow Q(s, o) + \eta \big[y - Q(s, o) \big]$.
        5. $s \leftarrow s'$.

Output: $Q \approx Q^*_{\mathcal{O}}$ and the greedy policy over options.

The target y in step 3 is a sample of the right-hand side of (20.3): G is a sample of $R_o(s)$, and γ^k times the value at s' is a sample of the discounted transition model applied to $\max_{o'} Q$. For this reason SMDP Q-learning converges to $Q^*_{\mathcal{O}}$ under the same conditions as one-step Q-learning (every available pair (s, o) updated infinitely often, step sizes satisfying the Robbins–Monro conditions, bounded rewards); the argument of Section 6.9 applies because the SMDP backup is a contraction.

A single update looks like this. In the deterministic four-rooms world the agent stands in S with $Q(\mathrm{S}, o) = 0.1$ for the option $o =$ "go to the hallway on the right", and $\max_{o'} Q$ in that hallway is currently 0.4. It runs the option, which takes $k = 7$ steps and collects no reward, so the target is $y = 0.9^7 \times 0.4 \approx 0.191$, and with $\eta = 0.5$ the new value is $0.1 + 0.5\,(0.191 - 0.1) \approx 0.146$. A one-step learner would need seven updates, in seven successive states, to move the same information over the same distance.

Figure 20.4 compares three learners on the slippery four-rooms world: Q-learning with primitive actions, SMDP Q-learning over the primitive actions and the hallway options, and the intra-

option method of the next subsection. All use $\eta = 0.25$, $\varepsilon = 0.1$ and initial values of zero. With options, the first ten episodes take about 100 steps on average instead of about 500. Initially all values are zero, so the agent chooses at random; random options carry it from room to room, and once the hallway next to the goal has a value, the next episode's backups spread it a room at a time. After about 40 episodes all three learners behave alike, at about 40 steps per episode (the slips and the 10% exploration keep it above the noise-free 16). Options helped the agent to *find* and *propagate* the reward; they did not change the final policy, which uses the primitive actions near the goal anyway.

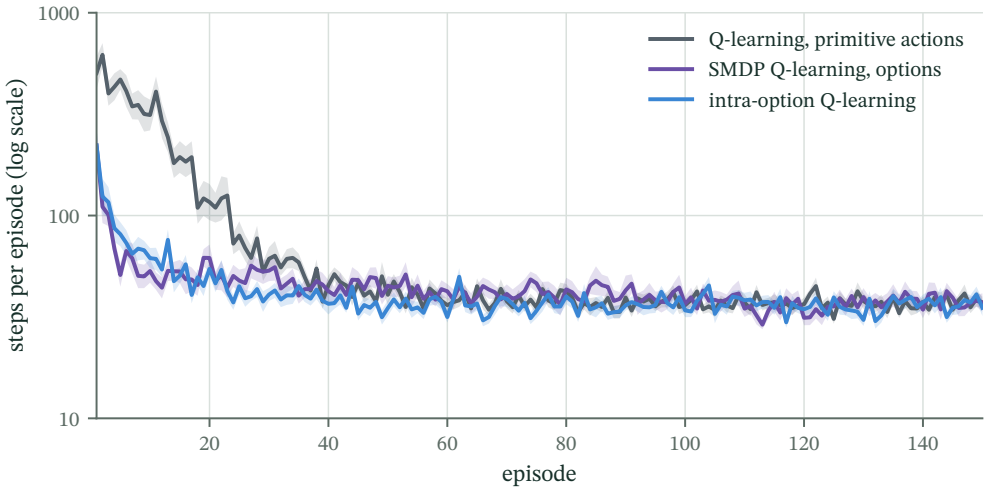


Figure 20.4. Steps per episode in the slippery four-rooms world for Q-learning with primitive actions, SMDP Q-learning over primitive actions and hallway options, and intra-option Q-learning (mean of $\log_{10}$ steps over 20 runs; the bands are ± 1 standard error). Options make the early episodes about five times shorter; after about 40 episodes the three learners are indistinguishable.

A TIME LIMIT IS NOT THE END OF AN OPTION

Suppose episodes are cut off after a fixed number of steps, and the cut falls while an option is running, in a state s' where the option would have continued. The tempting target $G + \gamma^k \max_{o'} Q(s', o')$ is wrong: it values a decision to switch options that the agent never made, and it overestimates whenever another option is better in s' than the running one. The correct target bootstraps from the value of letting the option go on, $G + \gamma^k \left[(1 - \beta_o(s'))\, Q(s', o) + \beta_o(s') \max_{o'} Q(s', o')\right]$. This is the option version of the truncation rule of Section 6.8. In the setting of the lab's exactness test, the tempting target leaves errors of 0.2 to 0.4, depending on the random seed, in values whose true size is at most 1.

LAB

Lab 20.1: Options, successor representations and hindsight (`labs/ch20_hierarchy.py`, about 45 minutes). Run `python3 labs/ch20_hierarchy.py`. Tasks 1–4 build the four-rooms world and its hallway options, compute an option's model exactly and check it by simulation, plan with SMDP value iteration, and compare the three learners of Figure 20.4. Look for the factor of about five in the length of the early episodes, and for the identical final behavior.

20.2.5 Intra-option learning ●●

SMDP Q-learning treats each option as an indivisible unit. It learns about an option only when that option is executed to completion, and only about the option that was executed; a ten-step option yields one update. Yet every primitive step carries information about every option that would have taken the same action in the same state. *Intra-option learning* (Sutton et al. 1999) exploits this. It is possible for Markov options because their values satisfy a Bellman equation at the level of single steps.

Let $U(s', o)$ be the value of arriving in s' while option o is running, before the termination decision is made. With probability $1 - \beta_o(s')$ the option continues and is worth $Q^*_{\mathcal{O}}(s', o)$; with probability $\beta_o(s')$ it stops, and the agent then chooses the best option available in s'. So

$$U(s', o) = \big(1 - \beta_o(s')\big)\, Q^*_{\mathcal{O}}(s', o) + \beta_o(s') \max_{o' \in \mathcal{O}_{s'}} Q^*_{\mathcal{O}}(s', o'),$$

and the option values satisfy the *intra-option Bellman equation*

$$Q^*_{\mathcal{O}}(s, o) = \sum_a \pi_o(a \mid s) \Big[r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, U(s', o) \Big]. \tag{20.4}$$

To see why, condition on the first action and the first next state. The option takes a with probability $\pi_o(a \mid s)$, earns $r(s, a)$ and moves to s'. If it then stops, the rest of the return is the value of the best option in s'; if it continues, the situation in s' is exactly that of starting o in s', because a Markov option's future behavior depends only on the current state. This is the step that fails for semi-Markov options, whose behavior in s' may depend on how long they have been running.

The sampled version of (20.4) is an off-policy, one-step update. Since it does not mention the option that was actually running, it can be applied to every option that is consistent with the observed action.

Algorithm 20.2 Intra-option Q-learning (Markov options with deterministic policies)

Input: a set of Markov options $\mathcal{O}$ with deterministic policies $\pi_o(s)$, step size η, discount γ.

1. Initialize $Q(s, o)$ for all s and $o \in \mathcal{O}_s$.
2. For every primitive transition (s, a, r, s') generated by any behavior:
   1. For every option o with $s \in \mathcal{I}_o$ and $\pi_o(s) = a$:
      1. $U \leftarrow 0$ if s' is terminal, else $U \leftarrow (1 - \beta_o(s'))\, Q(s', o) + \beta_o(s') \max_{o'} Q(s', o')$.
      2. $Q(s, o) \leftarrow Q(s, o) + \eta \big[r + \gamma U - Q(s, o) \big]$.

Output: $Q \approx Q^*_{\mathcal{O}}$.

For a primitive action viewed as an option, $\beta \equiv 1$ and the update is exactly Q-learning; intra-option Q-learning therefore contains one-step Q-learning as a special case. Sutton et al. (1999) prove that, for Markov options with deterministic policies, it converges to $Q^*_{\mathcal{O}}$ with probability

one for all options, whatever options are executed during learning, provided every primitive action is taken in every state infinitely often (with the usual step-size conditions). It makes one or more updates after every primitive step, where SMDP Q-learning makes one per completed option; in Figure 20.4 it is nearly as fast as SMDP Q-learning early on and slightly better later. With stochastic intra-option policies the same idea works with importance-sampling corrections, at the price of higher variance.

The step-level view also suggests changing the execution model. Under call-and-return, an option runs until its own termination function stops it. Sutton et al. (1999) show that *interrupting* a running option whenever $Q(s, o) < \max_{o'} Q(s, o')$, that is, whenever another option looks better in the current state, gives a policy that is at least as good as the uninterrupted one. With interruption, a planner can use coarse options and still recover fine-grained behavior when the options are not quite right.

20.3 Where options come from

Everything so far assumed that someone handed us good options. Where do they come from? There are four broad answers, and the rest of this section describes the main methods of each in ●● subsections.

By design. Engineers and roboticists usually write skills by hand: a grasp controller, a walking gait, a navigation routine. Hand-designed options encode knowledge that would be expensive to learn, and they are the most common form of hierarchy in practice. Their limitation is obvious: they are only as good as the designer's foresight.

From subgoals. Many tasks have *bottlenecks*, states that many good trajectories pass through, such as the hallways of the four rooms or a door in a building. An option that reaches a bottleneck is useful for many tasks. Methods in this family look for bottleneck states in experience or in the transition graph, and then learn options that reach them.

By optimization. Options can be parameterized, like policies, and their parameters adjusted by gradient ascent on the return of the task. The option-critic architecture does this end to end.

Without a task. An agent can practice before it is given any task, learning a repertoire of distinct, controllable behaviors, for example by maximizing how well its behavior reveals which skill it was executing. Skills learned this way can later serve as options.

Whatever the source, learned options are prone to two degenerate outcomes. They can *collapse in time*, terminating after every step (so that they are just primitive actions) or never terminating (so that one option does everything). And they can *collapse in content*, with several options learning the same behavior. Most methods add a regularizer, a cost or a diversity objective to prevent this, and judging whether the result is a good set of building blocks is still mostly done by inspection.

20.3.1 The option-critic architecture ●●

The *option-critic* architecture of Bacon et al. (2017) learns intra-option policies and termination functions by gradient ascent, together with a policy over options, using only the reward of the

task. Fix K options $\omega \in \{1, \dots, K\}$, available in every state, with intra-option policies $\pi_{\omega,\theta}(a \mid s)$ with parameters θ, termination functions $\beta_{\omega,\vartheta}(s)$ with parameters ϑ, and a policy over options $\pi_\Omega(\omega \mid s)$ (in practice ε-greedy with respect to the option values; for the theorems it is held fixed). Execution is call-and-return. Four value functions describe this system:

$$
\begin{aligned}
Q_\Omega(s,\omega) &= \sum_a \pi_{\omega,\theta}(a \mid s)\, Q_U(s,\omega,a),\\
V_\Omega(s) &= \sum_\omega \pi_\Omega(\omega \mid s)\, Q_\Omega(s,\omega),\\
Q_U(s,\omega,a) &= r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, U(\omega,s'),\\
U(\omega,s') &= \big(1-\beta_{\omega,\vartheta}(s')\big) Q_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s')\, V_\Omega(s').
\end{aligned}
$$

$Q_\Omega(s,\omega)$ is the value of running option ω from s; $Q_U(s,\omega,a)$ is the value of taking action a in s while running ω; and $U(\omega,s')$, as in the previous subsection, is the value of arriving in s' with ω still running. The *option advantage* is $A_\Omega(s,\omega) = Q_\Omega(s,\omega) - V_\Omega(s)$.

The pairs (s_t,ω_t) of state and running option form a Markov chain. Let $\mu_\Omega(s,\omega \mid s_0,\omega_0) = \sum_{t\ge 0} \gamma^t \Pr(s_t = s, \omega_t = \omega \mid s_0,\omega_0)$ be its discounted occupancy, the analogue of the discounted visitation of Definition 3.8 for state–option pairs.

Theorem 20.3 Intra-option policy gradient (Bacon, Harb and Precup 2017)

If the intra-option policies are differentiable in θ and π_Ω and the terminations do not depend on θ, then

$$
\frac{\partial Q_\Omega(s_0,\omega_0)}{\partial \theta} = \sum_{s,\omega} \mu_\Omega(s,\omega \mid s_0,\omega_0) \sum_a \frac{\partial \pi_{\omega,\theta}(a \mid s)}{\partial \theta}\, Q_U(s,\omega,a).
$$

Proof. Differentiate $Q_\Omega(s,\omega) = \sum_a \pi_{\omega,\theta}(a \mid s)\, Q_U(s,\omega,a)$ with the product rule:

$$
\frac{\partial Q_\Omega(s,\omega)}{\partial \theta} = \sum_a \frac{\partial \pi_{\omega,\theta}(a \mid s)}{\partial \theta}\, Q_U(s,\omega,a) + \sum_a \pi_{\omega,\theta}(a \mid s)\, \gamma \sum_{s'} P(s' \mid s,a)\, \frac{\partial U(\omega,s')}{\partial \theta}.
$$

The terminations do not depend on θ, so $\partial U(\omega,s')/\partial\theta = \sum_{\omega'} \big[(1-\beta_\omega(s'))\,\mathbb{1}\{\omega' = \omega\} + \beta_\omega(s')\,\pi_\Omega(\omega' \mid s')\big]\, \partial Q_\Omega(s',\omega')/\partial\theta$. The bracket is the probability that the option running in s' is ω', given that ω was running when s' was reached. The second term is therefore γ times the expected derivative at the next state–option pair of the chain. Unrolling the recursion, exactly as in the proof of the policy gradient theorem (Theorem 11.1), sums the first term over all future pairs with weights $\gamma^t \Pr(s_t = s, \omega_t = \omega)$, which is the claim. ∎

Theorem 20.4 Termination gradient (Bacon, Harb and Precup 2017)

If the termination functions are differentiable in ϑ and the intra-option policies and π_Ω do not depend on ϑ, then

$$
\frac{\partial U(\omega_0,s_1)}{\partial \vartheta} = -\sum_{s',\omega} \mu_\Omega(s',\omega \mid s_1,\omega_0)\, \frac{\partial \beta_{\omega,\vartheta}(s')}{\partial \vartheta}\, A_\Omega(s',\omega),
$$

where now $\mu_\Omega(s', \omega \mid s_1, \omega_0) = \sum_{t \geq 0} \gamma^t \Pr(s_{t+1} = s', \omega_t = \omega \mid s_1, \omega_0)$ weights the pairs (state reached, option that was running when it was reached).

The proof is the same unrolling, applied to $U(\omega, s') = Q_\Omega(s', \omega) - \beta_{\omega,\vartheta}(s') A_\Omega(s', \omega)$: the derivative of β appears multiplied by $-A_\Omega$, and the remaining terms pass the derivative on to the next arrival state. Both theorems are easy to check numerically, because on a finite MDP every quantity in them can be computed exactly by solving linear systems; the tests of this chapter compare both formulas with finite differences on random MDPs and find agreement to about 10^{-10}.

The termination gradient has a clear meaning. Gradient ascent *raises* the probability of terminating ω in s' when $A_\Omega(s', \omega) < 0$, that is, when ω is worse in s' than the average option, and lowers it when ω is better than average. This is the interruption rule of Section 20.2.5 made smooth.

Algorithm 20.3 Option-critic (tabular or linear version, after Bacon, Harb and Precup 2017)

Input: step sizes η (critic), η_θ, η_ϑ (actor), exploration rate ε, termination margin $\xi \geq 0$, discount γ.

1. Initialize $Q_U(s, \omega, a)$, θ, ϑ; observe s; choose ω ε-greedily with respect to $Q_\Omega(s, \cdot)$.
2. Repeat for every step:
   1. Take $a \sim \pi_{\omega,\theta}(\cdot \mid s)$; observe r and s'.
   2. Critic: $\delta \leftarrow r + \gamma\, U(\omega, s') - Q_U(s, \omega, a)$ (with $U = 0$ if s' is terminal); $Q_U(s, \omega, a) \leftarrow Q_U(s, \omega, a) + \eta\, \delta$.
   3. Intra-option policy: $\theta \leftarrow \theta + \eta_\theta \nabla_\theta \log \pi_{\omega,\theta}(a \mid s)\, Q_U(s, \omega, a)$.
   4. Termination: $\vartheta \leftarrow \vartheta - \eta_\vartheta \nabla_\vartheta \beta_{\omega,\vartheta}(s') \big(A_\Omega(s', \omega) + \xi\big)$.
   5. With probability $\beta_{\omega,\vartheta}(s')$, choose a new ω ε-greedily in s'.
   6. $s \leftarrow s'$.

Steps 3 and 4 are stochastic versions of the two theorems: the sampled pairs follow the occupancy μ_Ω (up to the usual missing discount factor, see Section 11.9), and the score-function form $\nabla \log \pi \cdot Q_U$ replaces $\sum_a \nabla \pi \cdot Q_U$ as in (11.4). The critic can use Q_U itself, as here, or learn Q_Ω by intra-option Q-learning and estimate Q_U from it. With deep networks, the same scheme learned options on Atari games (Bacon et al. 2017).

In practice, option-critic often learns options that terminate after very few steps (Harb et al. 2018). Nothing in the objective rewards long options: when two options are about equally good, terminating costs nothing, and the termination gradient follows whichever looks marginally better. The margin $\xi > 0$ in step 4 makes termination slightly costly. Harb et al. (2018) interpret it as a *deliberation cost*, the price of making a decision, and derive it as a penalty in the objective; they report that with it the options last longer and are more interpretable, and that performance on Atari games improves.

20.3.2 Bottlenecks and eigenoptions ●●

The oldest idea for discovering options is to look for subgoals. McGovern and Barto (2001) searched experience for states that appear often on successful trajectories and rarely on unsuccessful ones; Şimşek and Barto (2008) proposed the *betweenness* of a state in the transition graph, the fraction of shortest paths between other states that pass through it, as a measure of how useful it is as a subgoal. In the four-rooms graph, all eight cells with the highest betweenness are hallways or cells next to a hallway. Once subgoals are found, an option for each is learned with a pseudo-reward for reaching it, by any of the methods of this book. *Skill chaining* (Konidaris and Barto 2009) builds options backward from the goal in continuous domains: the first option reaches the goal, its initiation set becomes the target of the next option, and so on.

A second family is *spectral*. Machado et al. (2017) define *eigenoptions*: for each of the first few eigenvectors e of the graph Laplacian of the environment (later work uses the eigenvectors of the successor representation instead, Section 20.6), the option that maximizes the intrinsic reward $e(s') - e(s)$, that is, the option that climbs the eigenvector as fast as possible, stopping where it can climb no more. The smooth eigenvectors vary slowly within well-connected regions and change sign across bottlenecks (Figure 20.6), so eigenoptions travel between regions and help exploration. Their computation and the reasons they work are in Section 20.6.1.

20.3.3 Skills from mutual information ●●

Can an agent learn useful skills before it is given any task? One answer, developed in *variational intrinsic control* (Gregor et al. 2016) and in *DIAYN* ("diversity is all you need", Eysenbach et al. (2019)), is to ask for skills that are *distinguishable*. Draw a skill z from a fixed distribution $p(z)$, say uniform over K skills, and run the skill-conditioned policy $\pi_\theta(a \mid s, z)$ for an episode. Good skills visit different states, so that the state reveals the skill. DIAYN maximizes

$$\mathcal{F}(\theta) = I(S; Z) + \mathcal{H}(A \mid S, Z),$$

the mutual information between states and skills plus the entropy of the actions given the state and skill. The first term asks for distinguishable skills; the second asks each skill to act as randomly as possible while remaining distinguishable, which makes skills robust and spread out.

The mutual information involves the unknown posterior $p(z \mid s)$. A learned *discriminator* $q_\phi(z \mid s)$ gives a lower bound:

$$\begin{aligned} I(S; Z) = \mathcal{H}(Z) - \mathcal{H}(Z \mid S) &= \mathbb{E}_{z\sim p,\, s\sim\pi_z}\big[\log p(z \mid s) - \log p(z)\big] \\ &\geq \mathbb{E}_{z\sim p,\, s\sim\pi_z}\big[\log q_\phi(z \mid s) - \log p(z)\big], \end{aligned}$$

because the gap is an expected Kullback–Leibler divergence, $\mathbb{E}_s\big[D_{\mathrm{KL}}(p(\cdot \mid s) \,\|\, q_\phi(\cdot \mid s))\big] \geq 0$. The discriminator is trained to classify skills from states, which tightens the bound, and the policy is trained with the *pseudo-reward* $r_z(s) = \log q_\phi(z \mid s) - \log p(z)$ by a maximum-entropy method such as soft actor-critic (Algorithm 13.3), which takes care of the entropy term. The pseudo-reward is high in states where the discriminator is confident about the current skill. In simulated locomotion tasks, Eysenbach et al. (2019) report skills such as running, jumping and flipping that emerge without any task reward, and that can serve as options for a later task.

Distinguishable is not the same as useful. A skill can be easy to recognize because it does something irrelevant, and with limited capacity the agent may spend it on distinctions the later task does not care about. Variants condition the objective on predictable dynamics, as in DADS (Sharma et al. 2020), so that skills are also easy to plan with. Whether unsupervised skills make later learning faster, in general and not only on benchmarks designed for them, is an open question that Section 20.7 returns to.

20.4 Feudal and goal-conditioned hierarchies

An option tells the agent *which behavior* to run. A second way to build a hierarchy is to let a higher level decide *what to achieve* and a lower level work out *how*. The idea is old. In *feudal reinforcement learning* (Dayan and Hinton 1993), a hierarchy of managers navigates a maze at several resolutions: each manager gives commands to the sub-manager below it ("go north in your region") and receives commands from the one above. Two principles organize the system. *Reward hiding*: a sub-manager is rewarded for doing what its manager asked, whether or not this pleases the manager's own manager. *Information hiding*: each level sees the world only at its own resolution and knows nothing of the commands above its manager. Each level then solves a smaller, simpler problem.

Modern versions share a common structure. Every c steps, a *manager* (high-level policy) chooses a *goal* g in some goal space. In between, a *worker* (low-level policy) $\pi_{\text{lo}}(a \mid s, g)$ acts to achieve it, trained with an *intrinsic reward* for progress toward g. The manager is trained on the task's reward, summed over its c-step decisions; from its point of view, each decision is a temporally extended action, and its problem is an SMDP with duration c. The methods differ in what a goal is and in how they cope with the difficulties this creates.

- h-DQN (Kulkarni et al. 2016) used hand-specified goals, such as "reach the key" for objects detected in the image, a Q-learning meta-controller that chooses goals, and a controller rewarded for reaching them. With such goals it made progress in the first room of the Atari game Montezuma's Revenge, where the flat DQN of the time scored almost nothing.
- FeUdal Networks (FuN, Vezhnevets et al. (2017)) let the manager choose a *direction* g_t in a learned latent state space, at a lower temporal resolution. The worker receives the intrinsic reward $\frac{1}{c}\sum_{i=1}^{c} d_{\cos}(z_t - z_{t-i}, g_{t-i})$, where z is the latent state and $d_{\cos}$ the cosine similarity: it is rewarded for moving the latent state in the directions the manager asked for. The authors reported clear gains over a strong recurrent baseline on Atari games that need long-term credit assignment.
- HIRO (Nachum et al. 2018) uses goals in the raw state space: g_t is a desired change of state, the worker's reward is $-\|s_t + g_t - s_{t+1}\|$, and between the manager's decisions the goal is shifted, $g_{t+1} = s_t + g_t - s_{t+1}$, so that the absolute target stays fixed. Both levels are trained off-policy from replay, with a correction described below. HIRO learned to steer a simulated four-legged robot through mazes and to push obstacles out of its way, tasks on which the flat agents in its comparisons failed.
- Hierarchical actor-critic (HAC, Levy et al. (2019)) stacks several levels of goal-conditioned policies and uses hindsight (introduced in the next section) at every level. Its authors

reported that hierarchies with three levels, trained jointly, learned faster than two-level and flat agents on several continuous-control tasks.

The central difficulty is *non-stationarity of the higher level*. The manager's experience records that "in state s I chose goal g, and c steps later I was in s' with reward R". But what happens after choosing g depends on the worker, and the worker is learning. Old experience therefore describes an old worker, and the manager's replay data go stale. HIRO corrects this by *relabeling the goal*: it replaces the stored g by the goal $\tilde{g}$ that would have made the worker's recorded actions most likely under its *current* policy, chosen among a few candidates such as $s_{t+c} - s_t$. HAC instead relabels the manager's action with the state the worker actually reached, as if the worker had been perfect, which makes the manager's data consistent with an optimal worker. Both are forms of hindsight.

Two classical frameworks complete the picture. In *MAXQ* (Dietterich 2000), a designer decomposes the task into a tree of subtasks (in the original "taxi" example: navigate to the passenger, pick up, navigate to the destination, put down), and the value function decomposes along the tree,

$$Q(i, s, a) = V(a, s) + C(i, s, a),$$

where $V(a, s)$ is the expected discounted reward of executing child subtask (or action) a in s until it finishes, and the *completion function* $C(i, s, a)$ is the expected discounted reward of finishing the parent task i afterward. The decomposition allows strong state abstraction (navigating to the passenger does not depend on the destination) and yields *recursively optimal* policies, in which each subtask is solved optimally given its children; these can be worse than the best policy consistent with the hierarchy. In *hierarchies of abstract machines* (HAMs, Parr and Russell (1998)), the designer writes a partial program with choice points, and the agent learns only what to do at the choice points; the composition of the program and the MDP is again an SMDP, which is the same insight as for options.

KEY IDEA

Options choose which behavior to run; goal-conditioned hierarchies choose which outcome to aim for. Either way, the higher level faces a semi-Markov decision problem whose transitions depend on the lower level, and if the lower level is still learning, the higher level's experience goes stale.

20.5 Goal-conditioned reinforcement learning

Many agents are asked to do one of many related things: a robot arm should put a block at whatever position a user specifies, a navigation agent should reach whatever room it is sent to. Instead of learning a separate policy for each target, we can learn one policy and one value function that take the goal as an input. Kaelbling (1993) studied this setting under the name *learning to achieve goals*, and showed that in tabular problems, experience gathered while pursuing one goal can update the values of all goals at once. Schaul et al. (2015) proposed *universal value function approximators* (UVFAs): a single network $Q(s, a, g; w)$ that takes the state and the goal as inputs and generalizes across both, including to goals never trained on.

Definition 20.5 Goal-conditioned MDP

A *goal-conditioned MDP* is an MDP together with a set of goals $\mathcal{G}$, a distribution $p(g)$ over goals, and a map $\phi_{\mathcal{G}} : \mathcal{S} \to \mathcal{G}$ that says which goal a state *achieves*. Each episode draws a goal $g \sim p$, and the reward depends on the goal: in the sparse version used here, $r_g(s, a, s') = 1$ if $\phi_{\mathcal{G}}(s') = g$, which also ends the episode, and 0 otherwise. A goal-conditioned policy $\pi(a \mid s, g)$ and value function $Q(s, a, g)$ take the goal as an additional input.

The goal can be the whole state (reach this exact configuration) or part of it (put the block here, whatever the arm does). In continuous spaces the indicator becomes "within distance ϵ of g".

The setting has one great difficulty and one great opportunity, and the *bit-flipping* task of Andrychowicz et al. (2017) shows both. The state is a vector of n bits, the goal is another vector of n bits, and action i flips bit i. Reaching the goal earns $+1$ and ends the episode; episodes are cut off after n steps. The optimal policy is trivial: flip the bits that differ. If d is the number of bits in which s and g differ (the *Hamming distance*), the optimal value of flipping bit a is $Q^*(s, a, g) = \gamma^{d'}$, where $d' = d - 1$ if bit a differs and $d' = d + 1$ otherwise. Yet a standard deep Q-learning agent (Algorithm 10.1) fails. Its exploration is random, and a random policy reaches a random goal within n steps in only about 2.7% of episodes for $n = 8$, 0.9% for $n = 10$ and 0.3% for $n = 12$. Almost every episode ends with reward 0, and an agent that never sees a reward has nothing to learn from.

The opportunity is that a failed episode is a successful one *for a different goal*. An agent that aimed at g but ended in s_T has just demonstrated how to reach s_T. *Hindsight experience replay* (HER, Andrychowicz et al. (2017)) stores each transition twice or more: once with the original goal, and again with goals that were actually achieved later in the same episode, with the rewards recomputed for those goals.

Algorithm 20.4 Hindsight experience replay (with an off-policy learner)

Input: an off-policy goal-conditioned learner (for example DQN), a replay buffer $\mathcal{D}$, the number k of relabeled goals per transition, a goal-sampling strategy.

1. For each episode:
   1. Draw a goal g and an initial state s_0.
   2. Act with the behavior policy (for example ε-greedy in $Q(s_t, \cdot, g)$) until the episode ends at time T; record (s_t, a_t, s_{t+1}) for $t < T$.
   3. For each $t < T$:
      1. Store $(s_t, a_t, r_g(s_t, a_t, s_{t+1}), s_{t+1}, g)$ in $\mathcal{D}$.
      2. For $j = 1, \dots, k$: draw a goal g' achieved in this episode (strategy "future": $g' = \phi_{\mathcal{G}}(s_{t'})$ with t' uniform in $\{t+1, \dots, T\}$); store $(s_t, a_t, r_{g'}(s_t, a_t, s_{t+1}), s_{t+1}, g')$, ending the relabeled episode if $r_{g'} = 1$.
   4. Perform a number of learning updates on minibatches drawn from $\mathcal{D}$.

Output: a goal-conditioned value function and its greedy policy.

Other strategies draw g' from the final state of the episode ("final"), from any state of the episode ("episode"), or from the whole buffer ("random"); Andrychowicz et al. (2017) found "future" with $k = 4$ to work best on their robotics tasks.

Example 20.2 Relabeling a failed episode by hand

Take $n = 3$ bits, start $s_0 = 000$, goal $g = 111$. The agent flips bit 1, then bit 3, then bit 1 again, visiting 100, 101 and 001, and the episode is cut off after three steps. All three rewards are 0: the episode teaches nothing about reaching 111.

Relabel with the "final" strategy, $g' = 001$. The three stored transitions become $(000 \to 100)$, $(100 \to 101)$ and $(101 \to 001)$ with goal 001 and rewards $0, 0, 1$: the last one is a success, and Q-learning's bootstrapping passes its value back to the first two. With the "future" strategy and one goal per transition, transition t is relabeled with the state reached at a uniformly chosen later step, and it earns reward 1 when that step is its own. The chances are $1/3$, $1/2$ and 1 for the three transitions, so the relabeled copy contains on average $1/3 + 1/2 + 1 \approx 1.83$ successful transitions, and $k = 4$ copies contain about 7.3, where the original contained none.

♦

Figure 20.5 shows the result on bit flipping with a goal-conditioned DQN that takes the concatenation (s, g) as input (one hidden layer, 16 episodes and 40 minibatch updates per epoch). With HER, the success rate of the greedy policy reaches 100% for $n = 8$ in about 20 epochs, and close to 100% for $n = 16$ in about 80. Without HER, the agent stays at the success rate of random behavior: about 3% for $n = 8$ and essentially zero for $n = 16$.

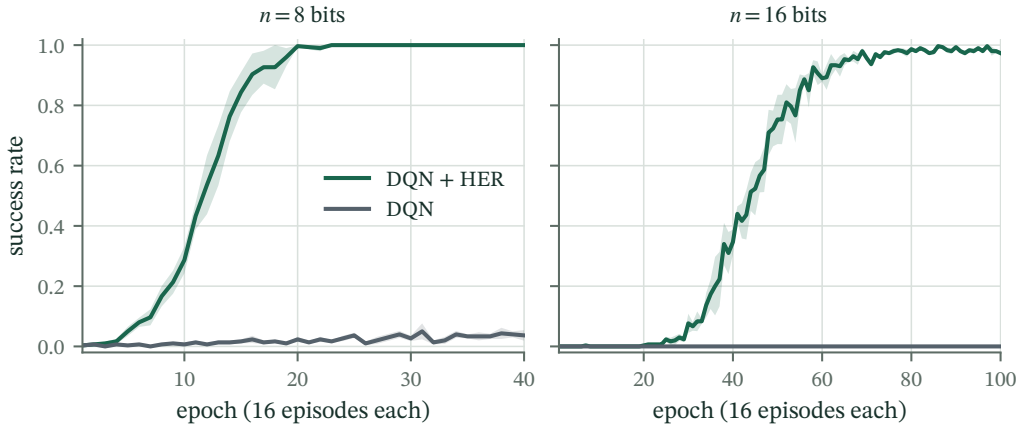


Figure 20.5. Success rate of the greedy policy on bit flipping with a goal-conditioned DQN, with and without hindsight experience replay, for $n = 8$ and $n = 16$ bits (mean of 3 runs; the bands are ± 1 standard error). Without HER the agent almost never sees a reward and does not improve; with HER it solves both problems.

Why is relabeling legitimate? A stored transition (s, a, s') is a sample of the environment's dynamics, which do not depend on the goal, and the reward for any goal can be recomputed from s'. An off-policy learner such as Q-learning can therefore learn $Q(\cdot, \cdot, g')$ from data collected while pursuing g, just as it learns about the greedy policy from ε-greedy data. There is one catch, and

it matters.

HINDSIGHT CAN BE BIASED WHEN TRANSITIONS ARE RANDOM

Relabeling picks the new goal *after seeing the outcome*. In a deterministic environment this is harmless, because s' is fixed by (s, a). In a stochastic one it biases the data toward lucky outcomes. Suppose one action leads to state A or state B with probability $1/2$ each, and the goal is A: the true value is $Q(s, a, A) = 1/2$. With goals drawn uniformly from $\{A, B\}$ and $k = 4$ "final" relabels per episode, the stored transitions with goal A have average reward $(1/2 + k)/(1 + k) = 0.9$, and Q-learning on the buffer converges to 0.9. Exercise 20.11 works through the example; corrections exist (Schramm et al. 2022).

Goal-conditioned learning and hierarchy fit together naturally. The worker of a feudal hierarchy is a goal-conditioned policy, and HAC applies hindsight to every level. Hindsight also suggests a simpler algorithm: *goal-conditioned supervised learning* (Ghosh et al. 2021) treats every relabeled trajectory as a demonstration of how to reach the goal it achieved, and trains the policy by imitation, without any value function.

LAB

Lab 20.1, Task 6 trains the goal-conditioned DQN on bit flipping with $n = 8$, with and without HER, in about three seconds. Watch the success rate with HER climb past 90% within about 15 epochs while the plain agent stays near 3%. The "going further" ideas push n up to see where HER starts to struggle.

20.6 The successor representation and successor features ●●

A value function mixes two things: where the policy takes the agent, and what is rewarded there. When the reward changes but the dynamics and the policy do not, as when the goal of the four rooms moves, we would like to keep the first part and recompute only the second. The *successor representation* (SR, Dayan (1993)) is that first part.

Definition 20.6 Successor representation

The *successor representation* of a policy π in a finite MDP is the matrix

$$M^\pi(s, s') = \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t \, \mathbb{1}\{s_t = s'\} \,\Big|\, s_0 = s\Big],$$

the expected discounted number of visits to s' when starting in s and following π (the visit at time 0 included). In episodic tasks the sum stops at the end of the episode, and M^π is indexed by the non-terminal states only.

Because $\mathbb{E}_\pi[\mathbb{1}\{s_t = s'\} \mid s_0 = s] = (P^\pi)^t(s, s')$, the SR is a geometric series of transition matrices:

$$M^\pi = \sum_{t=0}^{\infty} \gamma^t (P^\pi)^t = (I - \gamma P^\pi)^{-1}, \qquad V^\pi = M^\pi r^\pi. \tag{20.5}$$

This is the matrix that appeared in the solution $V^\pi = (I - \gamma P^\pi)^{-1} r^\pi$ of the Bellman equation (3.9), now given a name and an interpretation. In words: the value of a state is the reward of every state, weighted by how often (with discounting) the policy visits it.

Example 20.3 The successor representation of the Trail

Under the uniform policy on the Trail, from Start the hiker reaches Middle with probability 1/2 (one step later) and Goal with probability 1/4 (two steps later). With $\gamma = 0.9$ and the states ordered Start, Middle, Goal,

$$M^\pi = \begin{pmatrix} 1 & 0.45 & 0.2025 \\ 0 & 1 & 0.45 \\ 0 & 0 & 1 \end{pmatrix}, \qquad r^\pi = \begin{pmatrix} 0.5 \\ 1 \\ 4 \end{pmatrix}, \qquad M^\pi r^\pi = \begin{pmatrix} 1.76 \\ 2.8 \\ 4 \end{pmatrix}.$$

For example, $M^\pi(\text{Start}, \text{Goal}) = \gamma^2 \cdot \frac{1}{4} = 0.2025$, and $r^\pi(\text{Start}) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 = 0.5$ is the expected immediate reward in Start. The product reproduces the familiar values. If the reward of finish changed from 4 to 10, the new value of Start would be $0.5 + 0.45 \cdot 1 + 0.2025 \cdot 10 = 2.975$, read off without solving anything.

♦

The SR satisfies its own Bellman equation, $M^\pi = I + \gamma P^\pi M^\pi$: each row is a vector-valued value function whose "reward" is the indicator of the current state. It can therefore be learned by temporal differences from experience, without a model. After a transition from s to s',

$$M(s, \cdot) \leftarrow M(s, \cdot) + \eta \big[e_s + \gamma M(s', \cdot) - M(s, \cdot) \big],$$

where e_s is the indicator vector of s and $M(s', \cdot) = 0$ if s' is terminal. On the Trail, 20,000 episodes of this update with $\eta = 0.002$ give an estimate within a few hundredths of the exact M^π (Lab 20.1, Task 5).

Figure 20.6 shows the SR of the uniform random walk in the four rooms (no goal, $\gamma = 0.95$). Seen from a cell in the top-left room, 92% of the discounted visits stay in that room: the SR captures the geometry of the world, including its bottlenecks. Each row sums to $1/(1-\gamma) = 20$, the discounted number of time steps.

Successor features (Barreto et al. 2017) extend the idea to large state spaces. Suppose the reward is linear in known features $\phi(s, a) \in \mathbb{R}^d$: $r(s, a) = \phi(s, a)^\top w$ for a task vector w. The *successor features* of a policy are the expected discounted sum of future features,

$$\psi^\pi(s, a) = \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t \phi(s_t, a_t) \,\Big|\, s_0 = s,\ a_0 = a\Big], \qquad Q^\pi(s, a) = \psi^\pi(s, a)^\top w, \tag{20.6}$$

where the second identity follows from the linearity of expectation. With one-hot features, ψ^π is the SR of state–action pairs. The successor features satisfy $\psi^\pi(s, a) = \phi(s, a) + \gamma \mathbb{E}[\psi^\pi(s', a')]$ and can be learned by TD like any value function, one component per feature. Once they are known, the value of π for *any* task w is a dot product.

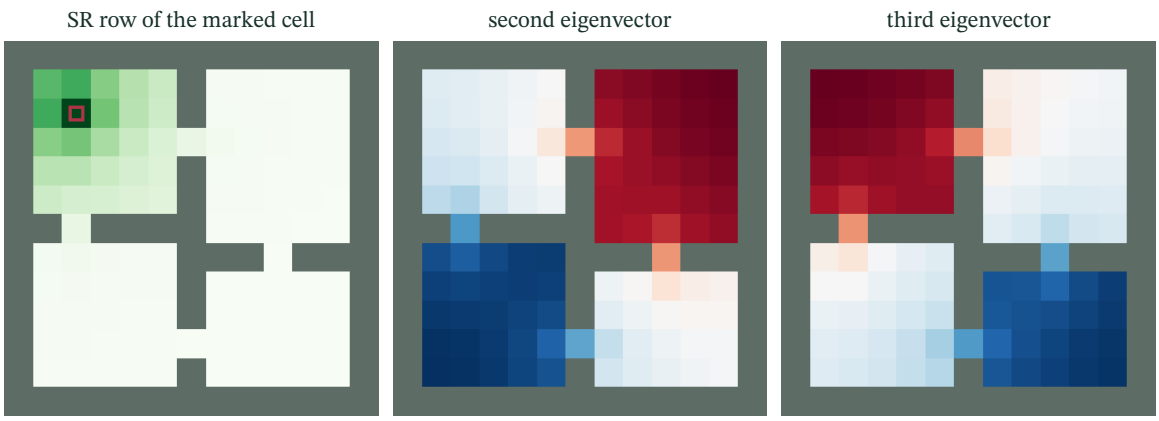


Figure 20.6. The successor representation of the uniform random walk in the four rooms with $\gamma = 0.95$. Left: the row of the marked cell, the expected discounted number of future visits to every cell; almost all of it stays in the starting room. Middle and right: the second and third eigenvectors of the SR (red positive, blue negative). They are nearly constant inside rooms and change sign across hallways, which is why they are useful for finding bottlenecks.

Evaluation of old policies on a new task is useful, but we want a *good* policy for the new task. *Generalized policy improvement* turns a library of old policies into a new one that is at least as good as each of them.

Theorem 20.7 Generalized policy improvement (Barreto et al. 2017)

Let $\pi_1, \dots, \pi_n$ be policies in an MDP, let $Q_{\max}(s,a) = \max_i Q^{\pi_i}(s,a)$, and let π be a deterministic policy with $\pi(s) \in \arg\max_a Q_{\max}(s,a)$. Then $Q^{\pi}(s,a) \geq Q_{\max}(s,a)$ for all s and a.

Proof. Let $\mathcal{T}^{\pi}$ be the Bellman expectation operator of π on action-value functions, $(\mathcal{T}^{\pi}Q)(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, Q(s', \pi(s'))$. For every i and every s', $Q_{\max}(s', \pi(s')) = \max_b Q_{\max}(s',b) \geq \max_b Q^{\pi_i}(s',b) \geq \sum_b \pi_i(b \mid s') Q^{\pi_i}(s',b)$. Hence

$$(\mathcal{T}^{\pi}Q_{\max})(s,a) \geq r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \sum_b \pi_i(b \mid s')\, Q^{\pi_i}(s',b) = Q^{\pi_i}(s,a)$$

for every i, so $\mathcal{T}^{\pi}Q_{\max} \geq Q_{\max}$. The operator $\mathcal{T}^{\pi}$ is monotone, so applying it repeatedly gives $(\mathcal{T}^{\pi})^m Q_{\max} \geq Q_{\max}$ for all m; and it is a γ-contraction (Theorem 4.3), so $(\mathcal{T}^{\pi})^m Q_{\max} \to Q^{\pi}$. Therefore $Q^{\pi} \geq Q_{\max}$. ■

With $n = 1$ this is the policy improvement theorem of Theorem 4.6. With successor features, $Q^{\pi_i} = \psi^{\pi_i \top} w$ is available for every new w at the cost of a dot product, so GPI gives an improved policy for a new task without any learning. If the Q^{π_i} are known only up to an error ϵ, the guarantee degrades gracefully, to $Q^{\pi} \geq Q_{\max} - 2\epsilon/(1-\gamma)$ (Barreto et al. 2017).

Example 20.4 Generalized policy improvement on the Trail

Give the Trail three features: $\phi = (1,0,0)$ for exiting at Start, $(0,1,0)$ for exiting at Middle, $(0,0,1)$ for finish. The usual rewards are $w = (1,2,4)$. Take two old policies: π_{exit}, which

exits wherever it can, and π_{go}, which goes all the way. Their successor features after go in Start are $\psi^{\pi_{\text{exit}}}(\text{Start}, \text{go}) = (0, \gamma, 0) = (0, 0.9, 0)$ (go, then exit at Middle) and $\psi^{\pi_{\text{go}}}(\text{Start}, \text{go}) = (0, 0, \gamma^2) = (0, 0, 0.81)$. Exiting gives $(1, 0, 0)$ under both.

Now a new task: exiting at Middle is worth 5, so $w' = (1, 5, 4)$. In Start, GPI compares $Q(\text{Start}, \text{exit}) = 1$ with $\max\{0.9 \cdot 5,\ 0.81 \cdot 4\} = \max\{4.5,\ 3.24\} = 4.5$ and goes. In Middle it compares exiting, worth 5, with $\psi^{\pi_{\text{go}}}(\text{Middle}, \text{go})^\top w' = 0.9 \cdot 4 = 3.6$, and exits. The GPI policy, "go, then exit at Middle", earns 4.5 on the new task; the old policies earn 1 and 3.24. It combines the first decision of one old policy with the last decision of the other, and here it is optimal for the new task.

♦

The SR also connects reinforcement learning with neuroscience: Stachenfeld et al. (2017) argued that place cells in the hippocampus behave as if they encoded a predictive map like the SR, rather than the animal's current position alone.

20.6.1 Eigenoptions and the Laplacian ●●●

The eigenvectors in Figure 20.6 are not an accident. For the uniform random walk in a grid world where blocked moves leave the agent in place, P^π is symmetric: each move to a neighboring cell has probability 1/4, and the remaining probability stays on the diagonal. Then

$$I - P^\pi = \tfrac{1}{4}(D - A) = \tfrac{1}{4}L,$$

where A is the adjacency matrix of the grid, D the diagonal matrix of degrees, and L the *graph Laplacian*. Consequently $M^\pi = (I - \gamma P^\pi)^{-1} = \left((1-\gamma)I + \frac{\gamma}{4}L\right)^{-1}$ has the same eigenvectors as L, and its largest eigenvalues belong to the Laplacian's smallest, the smoothest functions on the graph. These are the *proto-value functions* of Mahadevan and Maggioni (2007), proposed as basis functions for value-function approximation (Section 9.2). In the four rooms the random walk has four eigenvalues close to 1 (1, 0.994, 0.993 and 0.986), one per room, and then a gap to 0.929: the spectrum reveals that the world consists of four weakly connected parts.

Eigenoptions (Machado et al. 2017) use the smooth eigenvectors as intrinsic rewards, as described in Section 20.3.2. In general MDPs the transition matrix is not symmetric, and the SR's eigenvectors are the natural generalization; Machado et al. (2023) develop the SR as a common tool for discovering options (eigenoptions and covering options), for transferring with them (successor features and GPI) and for exploring. For large or continuous state spaces the eigenvectors must be learned; Wu et al. (2019b) give a scalable objective for learning Laplacian representations with neural networks.

RESEARCH NOTE

Open questions: Which spectral objects (eigenvectors of the SR, of the Laplacian, of a learned representation) give the most useful options for a given family of tasks, and can that be predicted before learning? How should successor features be chosen when the rewards of future tasks are not linear in known features? How do eigenoptions behave when the policy that defines the SR changes during learning?

20.7 What hierarchy has and has not delivered ●●●

The record of hierarchical reinforcement learning is uneven. Hand-designed hierarchies are everywhere: robots run libraries of skills, game agents use behavior trees, and engineers decompose tasks routinely. Goal-conditioned policies and hindsight relabeling are standard in robot learning. Learned hierarchies, on the other hand, have produced convincing gains on particular benchmarks (FuN on hard Atari games, HIRO and HAC on simulated locomotion and navigation, option-critic on small domains) without becoming a default tool in the way that DQN or PPO have. Most landmark results of deep reinforcement learning, in board games, video games and language models, used flat policies, with temporal structure supplied by memory, by action repetition, or by search.

Why does hierarchy help when it does? Nachum, Tang, et al. (2019) tested the usual explanations on locomotion, navigation and manipulation tasks where hierarchical agents had outperformed flat ones. They report that most of the benefit came from *better exploration*, not from easier learning over shorter horizons or from the hierarchical structure itself, and that flat agents given exploration strategies inspired by hierarchy were competitive. The small experiment of Figure 20.4 agrees in miniature: options shortened the early, exploratory episodes and did nothing for the final policy.

Theory gives a similar, nuanced picture. Good options provably help: they reduce the number of iterations that value iteration needs (Mann and Mannor 2014), they can lower the regret of exploration in a way that depends on how the options shape the state space (Fruit and Lazaric 2017), and repeated sub-structure can make both planning and learning more efficient (Wen et al. 2020). But the benefit depends on the options, and finding the best ones is hard: Jinnai et al. (2019) show that choosing a set of options that minimizes the planning time of value iteration is NP-hard. A bad set of options can also slow learning down, by multiplying the choices at every decision without shortening the paths to reward.

The open problems are therefore mostly about *discovery*: what makes a set of options good for a family of tasks, and how to learn it from experience at scale; how temporal abstraction should be paired with *state* abstraction, so that each skill sees only what it needs, as in MAXQ or feudal RL; how to keep the levels of a learned hierarchy from destabilizing each other; and how to build and use *models* of skills for planning, a direction that Sutton et al. (2023) pursue with subtasks designed so that their solutions remain useful for the main task.

Hierarchy has returned in a new form with agents built on large language models. A language model can act as a high-level policy that chooses among skills described in words. SayCan (Ahn et al. 2022) combined a language model's estimate of which robot skill would help with an instruction and a learned value function's estimate of whether the skill could succeed in the current state. Voyager (G. Wang et al. 2023) let a language model write and store skills as programs in the game Minecraft and reuse them for later tasks. A tool call in a language-model agent behaves much like an option: it starts a temporally extended procedure whose result returns control to the high level. Training such agents with reinforcement learning (Chapter 22) raises the questions of this chapter again: how to assign credit across turns and sub-procedures of very different lengths, whether to discount by time, by tokens or by decisions, and when a sub-procedure should stop.

RESEARCH NOTE

Open questions: Can options be discovered that transfer across many tasks, rather than serving one benchmark? Is there a measure of the quality of an option set that predicts its benefit before learning? When a language model proposes the skills, which parts of the classical theory (SMDP values, intra-option learning, interruption) still apply, and what replaces them when the skills are open-ended text or programs?

20.8 Summary

- An *option* is a policy with an initiation set and a termination function. Primitive actions are one-step options, so options extend the agent's repertoire without replacing it.
- An MDP with options is a *semi-Markov decision process*. An option is summarized by its multi-time model: the expected discounted reward it collects and the discounted distribution of where it stops. The SMDP Bellman equation has the same shape as the ordinary one, with γ^k for an option of duration k.
- Options never beat the optimal policy, and with the primitive actions included they match it; their value lies in speed. In the four rooms, value spreads a room per sweep, and early learning episodes are about five times shorter.
- SMDP Q-learning updates an option's value when it stops; intra-option Q-learning (●●) learns about every consistent option after every step. A time limit that cuts an option off must bootstrap from the option's own continuation value.
- Options can be designed, built around bottlenecks, learned by gradient ascent (the option-critic, with the intra-option policy gradient and termination gradient theorems, ●●), or learned without a task by making skills distinguishable (DIAYN, ●●).
- Feudal and manager–worker hierarchies let a higher level set goals for a lower one; the higher level's data go stale as the lower level learns, which methods like HIRO and HAC correct by relabeling.
- Goal-conditioned value functions serve many goals at once. Hindsight experience replay relabels failures as successes for the goals actually reached; it solves sparse-reward problems like bit flipping, and it is exact for deterministic transitions but biased for stochastic ones.
- (●●) The successor representation $M^\pi = (I - \gamma P^\pi)^{-1}$ separates dynamics from rewards: $V^\pi = M^\pi r^\pi$. Successor features make the values of old policies available for any linear reward, and generalized policy improvement combines them into a policy at least as good as each.
- (●●●) Hierarchy's benefits are real but mostly come from exploration; discovering good options remains open, and the ideas are returning in language-model agents that call skills and tools.

The next chapter brings other agents into the world, where the environment itself learns and adapts.

20.9 Further Reading

The options framework is laid out in Sutton et al. (1999), which remains the best single reference: it defines options and multi-time models, proves that options over an MDP form an SMDP, and introduces intra-option learning and interruption; Precup's thesis (Precup 2000) develops the theory in more depth. The surveys of Barto and Mahadevan (2003) and Pateria et al. (2021) cover the classical and the deep-learning eras of hierarchical reinforcement learning. For the other classical frameworks, read Dietterich (2000) on MAXQ, whose taxi example and discussion of recursive versus hierarchical optimality are still instructive, Parr and Russell (1998) on hierarchies of abstract machines, and Dayan and Hinton (1993) for the feudal principles.

On learning options, Bacon et al. (2017) is short and readable and states the gradient theorems in full; Harb et al. (2018) explain why learned options collapse and how a deliberation cost helps. Konidaris and Barto (2009) and Şimşek and Barto (2008) represent the subgoal tradition, and Machado et al. (2017) the spectral one. Eysenbach et al. (2019) is the clearest entry point to skill discovery by mutual information.

For manager–worker architectures, Vezhnevets et al. (2017), Nachum et al. (2018) and Levy et al. (2019) are the main papers; Kulkarni et al. (2016) shows the idea with hand-specified goals. For goals, Kaelbling (1993) is the origin, Schaul et al. (2015) introduced universal value function approximators, and Andrychowicz et al. (2017) hindsight experience replay, with experiments on bit flipping and robot manipulation.

The successor representation goes back to Dayan (1993). Barreto et al. (2017) introduce successor features and GPI, and Barreto et al. (2020) give a broad, readable account of how they support fast transfer. Machado et al. (2023) is a recent synthesis of the SR's role in temporal abstraction, and Stachenfeld et al. (2017) connects it to the brain.

(●●●) For a critical view, read Nachum, Tang, et al. (2019) on where the benefits of hierarchy come from, Jinnai et al. (2019) on the hardness of finding good options, and Sutton et al. (2023) for a current proposal on what subtasks should be. Ahn et al. (2022) and G. Wang et al. (2023) show how skill hierarchies reappear when a language model is the high-level policy.

20.10 Exercises

Exercise 20.1 ● Options on the Trail

On the ordinary Trail with $\gamma = 0.9$, consider the option o_G "go to Goal" of Section 20.2.1 together with the two primitive actions. (a) Compute $R_{o_G}(s)$ and $P^{\gamma}_{o_G}(\mathsf{Goal} \mid s)$ for $s = \mathsf{Start}, \mathsf{Middle}$. (b) Write the SMDP Bellman equation (20.2) for Start and check that $V^*(\mathsf{Start}) = 3.24$ satisfies it. (c) Run SMDP value iteration from $V = 0$ by hand, with and without o_G. After how many sweeps does $V(\mathsf{Start})$ reach 3.24 in each case?

Exercise 20.2 ● Counting decisions in the four rooms

In the deterministic four-rooms world, (a) write the optimal route from S to G as a sequence of hallway options and primitive actions with as few decisions as possible, and give the duration of each; (b) compute $P_o^\gamma(h \mid \mathsf{S})$ for the first option, where h is its target hallway; (c) use the SMDP Bellman equation to show that the value of the route is 0.9^{15}.

Exercise 20.3 • An SMDP Q-learning update

In the slippery four-rooms world, the agent in S has $Q(\mathsf{S}, o) = 0.1$ for the option o "go to the hallway on the right", and the largest option value in that hallway is 0.4. It runs o, which this time takes 9 steps because of two slips, and collects no reward. (a) With $\eta = 0.5$ and $\gamma = 0.9$, compute the new $Q(\mathsf{S}, o)$. (b) Compare with the update after the 7-step run of Section 20.2.4. (c) Why is it correct that a slower run of the same option yields a smaller target?

Exercise 20.4 •• Deriving the SMDP Bellman equation

Let o be a Markov option started in s at time t, with random duration k, and let V^μ be the value of a policy over options μ under call-and-return execution. (a) Show that the return from time t can be written $G_t = \sum_{j=0}^{k-1} \gamma^j r_{t+j} + \gamma^k G_{t+k}$. (b) Using the fact that, given $s_{t+k} = s'$, the process after time $t+k$ behaves as if it started in s' (why does this hold?), show that $V^\mu(s) = \sum_o \mu(o \mid s)\big[R_o(s) + \sum_{s'} P_o^\gamma(s' \mid s)\, V^\mu(s')\big]$. (c) Deduce (20.2) for the optimal value. (d) Show that $\sum_{s'} P_o^\gamma(s' \mid s) \le \gamma$, and conclude that the SMDP Bellman operator is a γ-contraction in the maximum norm.

Exercise 20.5 •• Multi-time models by linear algebra

Let o be a Markov option with policy π_o and termination β_o, and let P_π and r_π be the transition matrix and expected reward vector of π_o in the MDP. Write $B = \mathrm{diag}(\beta_o)$. (a) Show that the option model satisfies $R_o = r_\pi + \gamma P_\pi (I - B) R_o$ and $P_o^\gamma = \gamma P_\pi B + \gamma P_\pi (I - B) P_o^\gamma$, and solve both. (b) Show that $I - \gamma P_\pi (I - B)$ is invertible. (c) For an option whose duration is geometric with success probability p per step (as for "go" on the slippery Trail), show that $\mathbb{E}[\gamma^k] = p\gamma / (1 - (1-p)\gamma)$, and recover the numbers of Example 20.1.

Exercise 20.6 • A time limit inside an option

An option o is running when the episode is cut off by a time limit in a state s' where $\beta_o(s') = 0$. The partial discounted reward is $G = 0$ after $k = 3$ steps, $Q(s', o) = 0.3$, and the best option in s' has value $\max_{o'} Q(s', o') = 0.5$; $\gamma = 0.9$. (a) Compute the correct target and the tempting target of the pitfall in Section 20.2.4. (b) Explain in words why the tempting target overestimates. (c) Why does the problem not arise for intra-option Q-learning, or for one-step Q-learning with primitive actions?

Exercise 20.7 • The successor representation by hand

For the Trail policy with $p_0 = 0.8$ and $p_1 = 0.5$ and $\gamma = 0.9$: (a) write the transition matrix P^π among the non-terminal states and the reward vector r^π; (b) compute $M^\pi = I + \gamma P^\pi + \gamma^2 (P^\pi)^2$

and explain why the series stops; (c) compute $V^\pi = M^\pi r^\pi$ and check $V^\pi(\text{Start})$ against a direct calculation that lists the three paths of the Trail, as in Section 1.3; (d) use M^π to find the value of Start when the reward of finish is changed to 10.

Exercise 20.8 ●● TD learning of the successor representation

(a) Prove that $M^\pi = I + \gamma P^\pi M^\pi$ and that it is the unique solution of this equation. (b) Show that the expected TD target of the SR update in Section 20.6, $\mathbb{E}[e_s + \gamma M(s', \cdot) \mid s]$, is the s-th row of $I + \gamma P^\pi M$, so that the update is a stochastic approximation of a γ-contraction. (c) What are the row sums of M^π in a continuing task, and why? What are they in an episodic task?

Exercise 20.9 ●● Generalized policy improvement with successor features

Use the features and the two old policies of Example 20.4. (a) For the task $w = (3, 2, 4)$, find the GPI policy and its value at Start, and compare it with the values of the two old policies. (b) For $w = (1, 2, 4)$ and the single old policy π_{exit}, show that GPI reduces to one step of policy improvement, find the resulting policy, and compare its value with the lower bound $Q_{\max}$ that Theorem 20.7 guarantees. (c) Where in the proof of Theorem 20.7 is it used that all policies are evaluated on the same reward?

Exercise 20.10 ● Hindsight relabeling

In the episode of Example 20.2: (a) list the transitions stored with the "episode" strategy using the goal 100, with their rewards; (b) explain why relabeled transitions with goal 100 teach the agent something correct about the dynamics even though the agent never wanted to reach 100; (c) a classmate proposes to relabel goals with states *not* visited in the episode, chosen at random. What would the relabeled rewards be, and what would the agent learn from them?

Exercise 20.11 ●● When is hindsight relabeling unbiased?

(a) Consider a goal-conditioned MDP with deterministic transitions. Show that every relabeled transition (s, a, s', g') is consistent with the true dynamics and the true reward of goal g', so that Q-learning on a buffer of original and relabeled transitions has the true Q^* as a fixed point (assuming every pair (s, a, g) that matters is present). (b) Work through the stochastic example of the pitfall in Section 20.5: with goals uniform on $\{A, B\}$ and k "final" relabels, show that the average reward of stored transitions with goal A is $(1/2 + k)/(1 + k)$. (c) Explain which step of the argument in (a) fails in (b), and suggest one way to remove the bias.

Exercise 20.12 ●● The option-critic gradients

(a) Fill in the unrolling step of the proof of Theorem 20.3: write $\partial Q_\Omega(s_0, \omega_0)/\partial\theta$ as a sum over t of $\gamma^t \mathbb{E}\big[\sum_a \partial_\theta \pi_{\omega_t}(a \mid s_t)\, Q_U(s_t, \omega_t, a)\big]$. (b) Prove Theorem 20.4 in the same way,

starting from $U(\omega, s') = Q_\Omega(s', \omega) - \beta_\omega(s') A_\Omega(s', \omega)$. (c) For a sigmoid termination $\beta_\omega(s) = 1/(1 + e^{-\vartheta_{\omega,s}})$, write the stochastic update of $\vartheta_{\omega,s'}$ and explain in words when it makes the option stop more often.

Exercise 20.13 ●● The mutual-information lower bound

(a) Prove that $I(S; Z) \geq \mathcal{H}(Z) + \mathbb{E}[\log q(Z \mid S)]$ for every conditional distribution q, with equality if and only if $q(z \mid s) = p(z \mid s)$ wherever $p(s) > 0$. (b) With K skills drawn uniformly, what is the largest possible value of the DIAYN pseudo-reward $\log q(z \mid s) - \log p(z)$, and when is it attained? (c) Give an example of two skills that the discriminator can tell apart perfectly but that are equally useless for every task you can think of in the four-rooms world. What does this say about the objective?

Exercise 20.14 ● Programming: when are options enough?

Extend Lab 20.1. (a) Plan with the hallway options *alone* (no primitive actions) for a goal in the hallway between the two right-hand rooms, $\mathsf{G} = (7, 9)$, in the deterministic and in the slippery world, and compare $V(\mathsf{S})$ with $V^*(\mathsf{S})$. (b) Repeat with the goal inside the bottom-right room. (c) Explain the four results.

Exercise 20.15 ●●● Skills for language-model agents

A language-model agent solves software tasks by calling tools (run tests, search the code, edit a file), each of which runs for a variable time and returns a result. (a) Describe this agent as a policy over options: what are the initiation sets, intra-option policies and termination conditions? (b) Discuss how to discount and assign credit when tool calls take very different amounts of time, and compare discounting by decisions with discounting by elapsed time. (c) Which ideas of this chapter (SMDP Q-learning, intra-option learning, interruption, hindsight relabeling) could be useful for training such an agent with reinforcement learning, and what obstacles do you expect? Support your answer with references to the literature.

Exercise 20.16 ●●● Where do the benefits of hierarchy come from?

Read Nachum, Tang, et al. (2019). (a) Summarize the candidate explanations for the benefits of hierarchy that they test, and their conclusion. (b) Design an experiment, in the four-rooms world or a larger grid, that separates the contribution of exploration from that of faster value propagation, using the code of Lab 20.1. (c) Describe a setting in which you expect hierarchy to help for a reason other than exploration, and explain why.

CHAPTER 21

Multi-Agent Reinforcement Learning and Games

WHAT YOU WILL LEARN

- Explain why other learning agents make an agent's environment non-stationary, and distinguish cooperative, competitive and mixed settings.
- Compute best responses, Nash equilibria, minimax strategies (by linear programming) and correlated equilibria of small games.
- Define Markov games, solve a zero-sum Markov game with Shapley's value iteration, and learn it with minimax-Q.
- Explain why naive self-play and gradient play can cycle, and why averaging and no-regret learning converge in zero-sum games.
- Describe independent learning, relative overgeneralization, and the centralized-training methods VDN, QMIX, COMA, MADDPG and MAPPO, with their limits.
- Derive and implement counterfactual regret minimization and use it to solve Kuhn poker (●●).
- Summarize what is known about the complexity and dynamics of learning in games, and assess the landmark results of multi-agent reinforcement learning (●●●).

Every chapter so far had one learner. The environment could be complicated, stochastic or partly hidden, but it did not *care*: its transition probabilities stayed the same however the agent behaved. Many of the problems that matter most break this assumption. Cars at an intersection, traders in a market, robots in a warehouse, players in a game and, increasingly, software agents that negotiate or debate with each other all share their world with other decision makers, who learn and adapt too.

Multi-agent reinforcement learning (MARL) studies learning in this setting. Two things change. First, the objective is no longer obvious: when the outcome depends on what others do, "the best policy" is not well defined until we say what we assume about the others, and the answers come from *game theory*. Second, the learning problem becomes harder: from each agent's point of view the others are part of the environment, and since they are learning, the environment is *non-stationary*, which removes the foundation of the convergence results of Chapters 4 to 6. In return, the setting offers one of the most powerful ideas in the field, *self-play*, which produced superhuman programs for backgammon, Go, chess, poker and complex video games.

This chapter introduces the game theory needed to state multi-agent problems precisely and then develops the main families of methods. Its examples are small enough to solve exactly, so

that every equilibrium, value and convergence claim can be checked: matching pennies and rock-paper-scissors, a small zero-sum game between a striker and a goalkeeper, a cooperative game in which independent learners reliably fail, and Kuhn poker, a three-card game that is the standard test bed for algorithms for imperfect-information games.

Section 21.1 explains what changes when there are several learners. Section 21.2 covers the game-theoretic essentials: best responses, Nash equilibria, the minimax theorem, correlated equilibria and Markov games. Section 21.3 develops learning in zero-sum games: minimax-Q, self-play, and why naive gradient play cycles while averages converge. Section 21.4 treats teams of independent learners and their characteristic failures, and Section 21.5 the methods that train centrally and act separately. Section 21.6 (●●) turns to imperfect-information games and counterfactual regret minimization. Section 21.7 (●●●) surveys the theory of equilibrium computation and learning dynamics, and Section 21.8 (●●●) the landmark systems and the current frontier. The ● sections, Section 21.1 to 21.5, form a complete first course.

21.1 Many learners

Consider four situations. Two autonomous cars approach an intersection at the same time; each would like to go first, and both lose badly if both go. Two robots carry a table together; neither can lift it alone, and each is rewarded only if the table arrives. A poker program plays against a human; whatever one gains, the other loses. A fleet of delivery drones shares the airspace; each wants to finish its own deliveries quickly, and all want to avoid collisions. The first and last are *mixed* (partly common, partly conflicting interests), the second is *cooperative* (a shared reward), and the third is *competitive* (zero-sum).

In each case, an agent's best action depends on what the others do. A car should go if the other car yields and yield if it goes; there is no action that is best regardless. This is the first difference from single-agent RL: we must reason about *strategic* interaction, and the right notion of a good policy depends on the setting. In zero-sum games there is a compelling answer (play a strategy that guarantees the most against the worst opponent). In cooperative games the goal is clear (maximize the shared reward) but the agents must *coordinate*. In mixed games the standard answer, a Nash equilibrium, is a stability condition rather than a recommendation, and there may be many.

The second difference is *non-stationarity*. Suppose agent i treats the others as part of its environment. If they follow fixed policies π_{-i}, agent i faces an ordinary MDP whose transition probabilities average over the others' actions,

$$P_i(s' \mid s, a_i) = \sum_{a_{-i}} \pi_{-i}(a_{-i} \mid s)\, P(s' \mid s, a_i, a_{-i}),$$

where a_{-i} denotes the actions of all agents except i. But the others are learning, so π_{-i} changes, and with it P_i and the rewards. Experience gathered a thousand steps ago describes a world that no longer exists. Replay buffers go stale, value estimates chase a moving target, and the convergence theorems of Section 6.9 no longer apply.

A third difference is *credit assignment across agents*. When a team of robots receives one shared reward, each robot must work out how much of it its own actions earned, a problem on top of the usual credit assignment over time.

KEY IDEA

With several learners, "optimal" depends on what the others do, and each learner's environment changes as the others learn. Game theory supplies the solution concepts; multi-agent RL supplies algorithms that try to reach them from experience.

21.2 Game theory essentials

21.2.1 Normal-form games and best responses

The simplest multi-agent problem has one state and one decision for each player. A finite *normal-form game* has n players; player i has a finite set of actions $\mathcal{A}_i$ and a payoff function $u_i(a_1, \dots, a_n)$ that depends on everybody's actions. With two players we write the payoffs as matrices: the *row player* chooses a row i and receives A_{ij}, the *column player* chooses a column j and receives B_{ij}. A *mixed strategy* is a probability distribution over actions, x for the row player and y for the column player; with independent randomization the expected payoffs are $x^\top A y$ and $x^\top B y$. A *best response* to y is any strategy that maximizes $x^\top A y$; some pure action is always among the best responses, because $x^\top A y$ is an average of the payoffs $(Ay)_i$ of the pure actions.

Example 21.1 The prisoner's dilemma

Each of two players can cooperate (C) or defect (D). The row player's payoffs are

$$A = \begin{pmatrix} 3 & 0 \\ 5 & 1 \end{pmatrix} \quad \text{(rows and columns: C, D)},$$

and the game is symmetric: $B = A^\top$. Whatever the column player does, defecting pays the row player more ($5 > 3$ against C, $1 > 0$ against D); D is a *dominant* action, and by symmetry also for the column player. So both defect and each earns 1, although both would earn 3 by cooperating. The prisoner's dilemma is the classic illustration that individually rational behavior can be collectively poor, and its repeated version is a standard test of whether learning agents can discover cooperation.

♦

21.2.2 Nash equilibrium

When no action dominates, we look for strategies that are mutual best responses.

Definition 21.1 Nash equilibrium

A profile of mixed strategies $(x_1, \dots, x_n)$ is a *Nash equilibrium* if each x_i is a best response to the others: no player can increase their expected payoff by changing their own strategy while the others keep theirs.

Nash (1951) proved that every finite game has at least one Nash equilibrium in mixed strategies. Equilibria need not be pure, need not be unique, and need not be good for anybody, as the prisoner's dilemma shows. A Nash equilibrium is best read as a *stability* condition: if everybody expects everybody else to play it, nobody has a reason to deviate.

Mixed equilibria are found with the *indifference principle*: a player mixes between two actions only if both give the same expected payoff against the other's strategy, since otherwise shifting weight to the better one would pay. In *matching pennies* each player shows heads or tails; the row player wins 1 if the coins match and loses 1 otherwise. No pure profile is stable, because the loser always wants to switch. If the column player shows heads with probability y, the row player's payoffs are $2y - 1$ for heads and $1 - 2y$ for tails, which are equal only at $y = 1/2$; by symmetry $x = 1/2$, and the unique equilibrium is for both to flip a fair coin. *Rock-paper-scissors* works the same way, with the unique equilibrium $(1/3, 1/3, 1/3)$ for both players.

Example 21.2 Chicken: three equilibria

Two drivers head toward each other; each can dare (D) or swerve (C, "chicken"). The row player's payoffs (columns in the same order) are

$$A = \begin{pmatrix} 0 & 7 \\ 2 & 6 \end{pmatrix}, \qquad B = A^{\mathsf{T}}.$$

Both daring is a crash (0 each); a lone darer earns 7 and the swerver 2; both swerving earns 6 each. The pure profiles (D, C) and (C, D) are equilibria: the swerver would only crash by daring, and the darer would only lose by swerving. There is also a mixed one. If the column player dares with probability q, the row player earns $7(1-q)$ by daring and $2q+6(1-q)$ by swerving; these are equal at $q = 1/3$. By symmetry both dare with probability $1/3$, and each expects $7 \cdot \frac{2}{3} = \frac{14}{3} \approx 4.67$. Notice that the mixed equilibrium gives both players less than mutual swerving would, and that it crashes with probability $1/9$.

♦

21.2.3 Zero-sum games and the minimax theorem

A two-player game is *zero-sum* if $B = -A$: one player's gain is the other's loss. Here a cautious player has a clear goal: maximize the payoff that can be *guaranteed* against every opponent. The row player's guaranteed payoff with strategy x is $\min_y x^{\mathsf{T}} A y = \min_j (x^{\mathsf{T}} A)_j$ (the worst column), and the best guarantee is the *maximin* value $\max_x \min_y x^{\mathsf{T}} A y$. The column player, symmetrically, can hold the row player down to $\min_y \max_x x^{\mathsf{T}} A y$.

Theorem 21.2 Minimax theorem (von Neumann 1928)

For every finite two-player zero-sum game with payoff matrix A,

$$\max_x \min_y \; x^\top A y = \min_y \max_x \; x^\top A y,$$

where x and y range over mixed strategies. The common number is the *value* of the game, $\mathrm{val}(A)$; a pair (x^*, y^*) attaining it is a Nash equilibrium, and conversely every Nash equilibrium attains it.

The theorem (Neumann 1928) has strong consequences. A maximin strategy is *safe*: it earns at least $\mathrm{val}(A)$ against every opponent, clever or not. Equilibria are *interchangeable*: if (x, y) and (x', y') are equilibria, so are (x, y') and (x', y), so the players need not coordinate on which one to play. And the value and a maximin strategy can be computed by *linear programming*: the row player chooses x and a number v to

$$\text{maximize } v \quad \text{subject to} \quad \sum_i x_i A_{ij} \geq v \text{ for every column } j, \qquad \sum_i x_i = 1, \qquad x \geq 0. \tag{21.1}$$

In words: find the mixed strategy whose worst-case column payoff is as large as possible. The column player's minimax strategy is the solution of the dual program. Solving (21.1) for matching pennies gives $x = (1/2, 1/2)$ and $v = 0$, and for rock-paper-scissors $x = (1/3, 1/3, 1/3)$ and $v = 0$. When the row player has only two actions, the guaranteed payoff $\min_j [pA_{1j} + (1-p)A_{2j}]$ is a concave, piecewise-linear function of $p = x_1$, and its maximum can be found by checking the endpoints and the crossing points of the lines, which is what the function `value_2xn` of the lab does.

In general-sum games none of this holds: there are many equilibria with different payoffs, they are not interchangeable, and playing one's part of an equilibrium guarantees nothing if the others play a different one.

21.2.4 Correlated equilibria

In chicken, the mixed equilibrium is wasteful because the drivers randomize independently and sometimes crash. A traffic light does better. It draws a joint recommendation, tells each driver only their own part, and each driver is free to ignore it. Aumann (1974) called a distribution z over joint actions a *correlated equilibrium* if obeying is a best response for every player who expects the others to obey: for every player, every recommended action a and every alternative a',

$$\sum_{a_{-i}} z(a, a_{-i}) \left[u_i(a, a_{-i}) - u_i(a', a_{-i}) \right] \geq 0.$$

These are linear inequalities in z, so correlated equilibria form a convex polytope and can be found, optimized over, by linear programming, in time polynomial in the size of the game. Every Nash equilibrium is a correlated equilibrium (with z the product of the mixed strategies), but not conversely.

In chicken, the light that recommends (C, C), (D, C) and (C, D) with probability $1/3$ each is a correlated equilibrium. A driver told D knows that the other was told C, and daring (7) beats swerving (6); a driver told C thinks the other dares with probability $1/2$, and swerving ($\frac{1}{2}\cdot 2 + \frac{1}{2}\cdot 6 =$

4) beats daring ($\frac{1}{2}\cdot 0 + \frac{1}{2}\cdot 7 = 3.5$). Each driver expects 5, more than the 14/3 of the mixed Nash equilibrium, and nobody crashes. The linear program that maximizes the total payoff finds an even better one: (C, C) with probability 1/2 and each of the other two with 1/4, worth 5.25 to each driver. A slightly weaker notion, the *coarse correlated equilibrium*, only requires that no player prefers to ignore the recommendation altogether in favor of a fixed action; it will reappear in Section 21.7 as the natural outcome of no-regret learning.

21.2.5 Markov games

Normal-form games have no state. The multi-agent analogue of an MDP adds one.

Definition 21.3 Markov game

A *Markov game* (or *stochastic game*) with n players consists of a set of states $\mathcal{S}$; an action set $\mathcal{A}_i$ for each player; a transition probability $P(s' \mid s, a_1, \dots, a_n)$; a reward function $r_i(s, a_1, \dots, a_n)$ for each player; a discount factor γ; and an initial-state distribution. In every state, all players choose actions simultaneously; player i receives r_i, and the game moves to the next state. Each player's goal is to maximize the expected discounted sum of its own rewards.

The model goes back to Shapley (1953); Littman (1994) introduced it to reinforcement learning. It contains the earlier cases: with one player it is an MDP, with one state a repeated normal-form game, and with $r_1 = \cdots = r_n$ a *team* problem. When each player sees only its own observation, it becomes a *decentralized POMDP* (Oliehoek and Amato 2016), the standard model of cooperative multi-agent control. A policy for player i maps states to distributions over $\mathcal{A}_i$; with a joint policy $\pi = (\pi_1, \dots, \pi_n)$, each player has a value function $V_i^{\pi}(s)$, and a Nash equilibrium is a joint policy in which each π_i is a best response, in the MDP sense, to the others. For finite discounted Markov games, Nash equilibria in stationary (Markov) policies always exist (Fink 1964).

For two-player zero-sum Markov games ($r_2 = -r_1$) the theory is as clean as for MDPs. Write r for the reward of player 1, the maximizer, and a and o for the actions of the maximizer and the minimizer. In each state the players face a matrix game whose payoffs combine the immediate reward and the continuation values,

$$Q_V(s, a, o) = r(s, a, o) + \gamma \sum_{s'} P(s' \mid s, a, o)\, V(s'),$$

and the *Shapley operator* replaces $V(s)$ by the value of that matrix game, $(\mathcal{T}V)(s) = \operatorname{val}\big(Q_V(s, \cdot, \cdot)\big)$.

Theorem 21.4 Shapley's theorem (1953)

In a finite two-player zero-sum Markov game with $\gamma < 1$, the Shapley operator is a γ-contraction in the maximum norm. Its unique fixed point V^* is the value of the game: the stationary policies that play a maximin (respectively minimax) strategy of the matrix game $Q_{V^*}(s, \cdot, \cdot)$ in every state guarantee each player $V^*(s)$ against every opponent, from every state s.

Proof. The key fact is that the value of a matrix game moves by at most the largest change of a payoff: $|\operatorname{val}(A)-\operatorname{val}(B)| \le \max_{ij}|A_{ij}-B_{ij}| =: \delta$. Indeed, for all mixed x and y, $x^{\top}Ay \le x^{\top}By+\delta$, so $\min_y x^{\top}Ay \le \min_y x^{\top}By + \delta$ for each x, and maximizing over x gives $\operatorname{val}(A) \le \operatorname{val}(B) + \delta$; exchanging A and B gives the other inequality. For two value functions V and W, the payoffs of the matrix games in state s differ by $\gamma \sum_{s'} P(s' \mid s,a,o)(V(s') - W(s'))$, at most $\gamma\|V - W\|_\infty$ in absolute value, so $|(\mathcal{T}V)(s) - (\mathcal{T}W)(s)| \le \gamma\|V - W\|_\infty$. Banach's fixed-point theorem (Theorem 4.4) gives the unique fixed point. That the stage-game strategies guarantee V^* follows as in the single-agent case. Fix the maximizer's stationary maximin policy; the minimizer then faces an MDP. Each stage strategy guarantees $\operatorname{val}\big(Q_{V^*}(s,\cdot,\cdot)\big) = V^*(s)$ against every column, so that MDP's Bellman optimality operator maps V^* to a function that is at least V^*. By monotonicity and contraction, the minimizer's optimal value in this MDP, the least the maximizer can be held to by any policy of the minimizer, is at least V^*. The argument for the minimizer is symmetric. ∎

Shapley's value iteration, $V \leftarrow \mathcal{T}V$, therefore converges geometrically, as value iteration does for MDPs (Algorithm 4.3); the only new ingredient is that each backup solves a small matrix game, by linear programming or, with two actions, by the formula of Section 21.2.3. The theorem also says something important about policies: in zero-sum Markov games, the optimal policies are in general *stochastic*. Randomizing has no value in an MDP (Theorem 3.7), but against an adversary who would exploit any predictable choice it is essential.

Example 21.3 A striker and a goalkeeper

A striker attacks the goal from far away, from the edge of the box, or from close range. In each round the striker chooses to *shoot* or to *dribble* and, at the same moment, the goalkeeper chooses to *stay* on the line or to *rush* out. A goal is worth 1 to the striker and -1 to the keeper; the game is undiscounted and every attack ends within three rounds. The rules are in Table 21.1.

Table 21.1. The striker-and-goalkeeper game. Shooting ends the attack; dribbling past a keeper who stays moves the striker one step closer (from close range it is a tap-in); dribbling against a keeper who rushes either loses the ball or rounds the keeper for an empty-net goal.

Striker	Keeper	far	edge of box	close
shoot	stay	goal 5%	goal 15%	goal 30%
shoot	rush	goal 40%	goal 50%	goal 70%
dribble	stay	move to edge	move to close	goal 60%
dribble	rush	goal 30%	goal 30%	goal 20%

Solve it backward. From close range the stage game, with rows shoot and dribble and columns stay and rush, is $\left(\begin{smallmatrix} 0.3 & 0.7 \\ 0.6 & 0.2 \end{smallmatrix}\right)$. It has no pure saddle point: against a keeper who stays the striker should dribble, against one who rushes he should shoot, and the keeper's best reply flips accordingly. The indifference principle gives the striker's shooting probability p from $0.3p+0.6(1-p) = 0.7p + 0.2(1-p)$, so $p = 1/2$, and the value 0.45. At the edge of the box, dribbling past a keeper

who stays is worth $V(\text{close}) = 0.45$, so the stage game is $\left(\begin{smallmatrix} 0.15 & 0.5 \\ 0.45 & 0.3 \end{smallmatrix}\right)$, with $p = 0.3$ and value 0.36. From far away it is $\left(\begin{smallmatrix} 0.05 & 0.4 \\ 0.36 & 0.3 \end{smallmatrix}\right)$, with $p = 6/41 \approx 0.146$ and value $129/410 \approx 0.315$. The keeper's minimax strategy stays on the line with probability 0.244, 0.4 and 0.625 in the three states. The optimal striker shoots more the closer he gets, and keeps the keeper guessing everywhere.

♦

21.3 Learning in zero-sum games

21.3.1 Minimax-Q

If the game is unknown, each player can learn it from experience, much as Q-learning learns an MDP. Littman (1994) proposed *minimax-Q*: keep a value $Q(s, a, o)$ for every state and *pair* of actions, and replace the maximum in Q-learning's target by the value of the stage matrix game.

Algorithm 21.1 Minimax-Q (for the maximizing player of a zero-sum Markov game)

Input: step sizes η_t, exploration rate ε, discount γ.

1. Initialize $Q(s, a, o) = 0$ for all states and action pairs.
2. Repeat for every step, in state s:
   1. Compute a maximin strategy x_s of the matrix game $Q(s, \cdot, \cdot)$, by the linear program (21.1).
   2. With probability ε choose a uniformly at random; otherwise draw $a \sim x_s$.
   3. Observe the opponent's action o, the reward r and the next state s'.
   4. $V(s') \leftarrow \text{val}\big(Q(s', \cdot, \cdot)\big)$, or 0 if s' is terminal.
   5. $Q(s, a, o) \leftarrow Q(s, a, o) + \eta_t \left[r + \gamma V(s') - Q(s, a, o)\right]$.

Output: $Q \approx Q^*$ and the stationary policy $s \mapsto x_s$.

Minimax-Q learns the game, not a best response to the particular opponent it meets. The target does not depend on how the opponent behaves, as long as every action pair is tried in every state infinitely often; in this sense the method is off-policy with respect to both players. Littman and Szepesvári (1996) proved that minimax-Q converges to Q^* with probability one under the usual conditions of Theorem 6.1: bounded rewards, step sizes with $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$ for every triple (s, a, o), and every triple updated infinitely often. The proof is that of Q-learning, with the max operator replaced by the val operator, which is also a non-expansion.

On the striker-and-goalkeeper game, with both players acting at random and the attack starting in a random state, minimax-Q's values after 30,000 attacks are within about 0.01 of the exact values $(0.315, 0.36, 0.45)$, and its shooting probabilities are close to $(0.146, 0.3, 0.5)$. Compare an ordinary Q-learner that treats the keeper as part of the environment and trains against a keeper who acts at random. It learns the deterministic policy "dribble, dribble, shoot", which is the best reply to a random keeper. Against a keeper who knows this policy, it scores only 0.30 from every state, against the minimax guarantee of 0.315, 0.36 and 0.45: at close range the predictable

striker loses a third of his scoring chance. Littman's original experiments, on a small soccer game, made the same point: policies learned by ordinary Q-learning were much more exploitable by a challenger trained specifically against them than the policies learned by minimax-Q.

In general-sum Markov games, the natural extension replaces val by the payoff of some Nash equilibrium of the stage game. *Nash-Q* (Hu and Wellman 2003) does this, but its convergence requires strong conditions, essentially that every stage game met during learning has a global optimum or a saddle point and that the same kind is selected consistently; *friend-or-foe Q-learning* (Littman 2001) converges under weaker assumptions but to a Nash equilibrium only in games with purely cooperative or purely adversarial equilibria. Neither problem is an accident of these algorithms: in general-sum games, equilibria are many, they are hard to compute (Section 21.7), and an equilibrium of every stage game does not single out a sensible joint policy.

LAB

Lab 21.1: Games, equilibria and learning among several agents (`labs/ch21_marl.py`, about 45 minutes). Run `python3 labs/ch21_marl.py`. Task 1 computes the equilibria of this section by enumeration and linear programming; Task 3 solves the striker-and-goalkeeper game with Shapley's value iteration, learns it with minimax-Q and measures how much the naive Q-learner can be exploited.

21.3.2 Self-play

Minimax-Q needs to see the opponent's actions and to solve a matrix game at every step, which is impossible in large games. A simpler idea has been far more successful in practice: let the agent play against itself. In *self-play*, a single learner controls both sides (or plays against copies of itself), and each side treats the other as part of the environment. As the agent improves, so does its opponent, which produces an automatic curriculum: the opponent is always about as strong as the agent. TD-Gammon learned backgammon this way (Tesauro 1995), and AlphaGo Zero and AlphaZero learned Go, chess and shogi from self-play alone, combining it with tree search (Silver, Schrittwieser, et al. 2017; Silver et al. 2018). In two-player zero-sum games with perfect information, where a single deterministic policy can be optimal, self-play with strong search is remarkably effective.

Naive self-play can also go in circles. In rock-paper-scissors, a learner that best-responds to its current self cycles forever: rock is beaten by paper, which is beaten by scissors, which is beaten by rock. *Best-response dynamics* in matching pennies cycle through all four pure profiles in the same way. In richer games, the cycles take the form of strategies that beat the current opponent but lose to earlier ones, so that a later agent is not stronger in any absolute sense, only different. And an agent that only meets itself can overfit to its own quirks and be exploited by a different opponent.

The classical remedy is to respond to the opponent's *history*, not to its latest version. In *fictitious play* (Brown 1951) each player best-responds to the empirical frequency of the other's past actions. Robinson (1951) proved that in two-player zero-sum games the empirical frequencies converge to the set of equilibria; in rock-paper-scissors, after 10,000 rounds the frequencies are

within about 0.01 of $(1/3, 1/3, 1/3)$, while the actions themselves keep cycling, in runs that grow longer and longer. In general-sum games fictitious play need not converge: Shapley (1964) gave a 3×3 game in which it cycles forever. Large-scale systems use the same idea with neural networks: *fictitious self-play* and *neural fictitious self-play* (NFSP, Heinrich and Silver (2016)) train a best response by RL against the average of past policies, which they learn by supervised learning; *population-based* methods keep a league of past and present policies and choose opponents from it, as AlphaStar did (Section 21.8); and *policy-space response oracles* (PSRO, Lanctot et al. (2017)) alternate between solving the game restricted to the current population of policies and adding an RL best response to the solution, generalizing the double-oracle algorithm of McMahan et al. (2003).

21.3.3 Why naive gradient play cycles

Gradient-based learners face the same problem in a continuous form. In matching pennies, let x and y be the probabilities that the row and column players show heads. The row player's expected payoff is $u(x, y) = (2x - 1)(2y - 1)$; the row player ascends it and the column player descends it. *Simultaneous gradient play* updates

$$x \leftarrow x + \eta \frac{\partial u}{\partial x} = x + 2\eta\,(2y - 1), \qquad y \leftarrow y - \eta \frac{\partial u}{\partial y} = y - 2\eta\,(2x - 1),$$

with the results clipped to $[0, 1]$. In continuous time ($\eta \to 0$) the quantity $(2x - 1)^2 + (2y - 1)^2$ is conserved: the strategies circle the equilibrium $(1/2, 1/2)$ forever. With a finite step it is worse: writing $u = 2x - 1$ and $w = 2y - 1$, one step maps (u, w) to $(u + 4\eta w,\ w - 4\eta u)$, which multiplies $u^2 + w^2$ by exactly $1 + 16\eta^2$. The strategies spiral *outward* until they hit the boundary of the square, and then circle along it (Figure 21.1, left). Gradient play, the most natural learning rule of the earlier chapters, does not converge in the simplest game with a unique equilibrium.

The same figure shows the way out: the *time average* of the strategies converges to the equilibrium, even though the strategies themselves do not. This is no coincidence. A learner has *no regret* if, in the long run, it does as well as the best fixed action in hindsight. Formally, if it plays x^t against y^t for $t = 1, \dots, T$, its *average regret*

$$R^T = \max_x \frac{1}{T} \sum_{t=1}^{T} x^\top A y^t - \frac{1}{T} \sum_{t=1}^{T} (x^t)^\top A y^t$$

tends to zero (or below) whatever the opponent does. Projected gradient ascent with suitably decreasing step sizes has this property, and so does *regret matching* (Hart and Mas-Colell 2000), which plays each action with probability proportional to its positive cumulative regret,

$$x^{T+1}(a) = \frac{\max\big(\sum_{t \le T} [u(a, y^t) - u(x^t, y^t)],\ 0\big)}{\sum_b \max\big(\sum_{t \le T} [u(b, y^t) - u(x^t, y^t)],\ 0\big)},$$

or uniformly if no regret is positive; its average regret is at most of order $\sqrt{|\mathcal{A}|/T}$ times the range of the payoffs. In a zero-sum game, if both players have average regret at most ϵ, their average strategies form a 2ϵ-equilibrium: neither player can gain more than 2ϵ by deviating from its average strategy (Exercise 21.8). Regret matching in rock-paper-scissors illustrates both halves

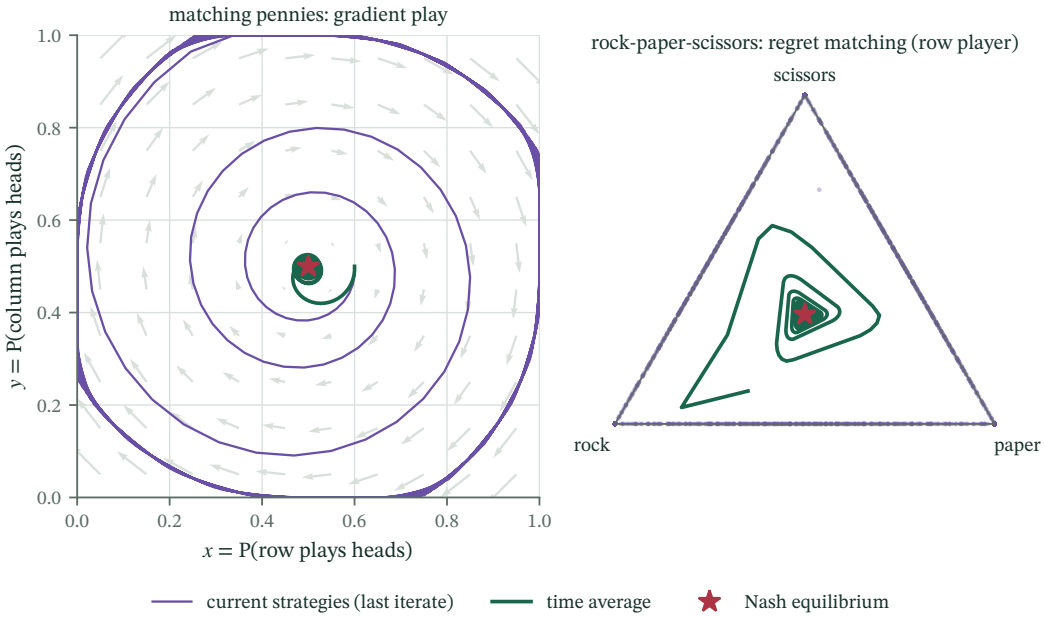


Figure 21.1. Learning dynamics in zero-sum games. Left: simultaneous gradient play in matching pennies ($\eta = 0.05$, starting at $x = 0.6$, $y = 0.5$); the arrows show the continuous-time vector field, which circles the equilibrium. The strategies (violet) spiral out to the boundary, but their time average (green) converges to the equilibrium. Right: regret matching in rock-paper-scissors; the current strategy of the row player (violet dots) jumps around the edges of the probability simplex, while the average strategy spirals into the equilibrium $(1/3, 1/3, 1/3)$.

of the story (Figure 21.1, right): its current strategy keeps jumping between the edges of the simplex, while the average converges.

Two lessons follow for multi-agent learning. When only the *average* behavior converges, the policy to deploy is the average policy, which is why fictitious play, NFSP and the counterfactual regret minimization of Section 21.6 all output averages. And when the *last iterate* should converge (for example, because averaging neural networks is awkward), the learning rule must change: variable step sizes that learn fast when losing and slowly when winning (*WoLF*, Bowling and Veloso (2002)), optimistic or extragradient updates, or regularization; Section 21.7 states what is known.

21.4 Independent learners and teams

The simplest way to build a multi-agent learner is to ignore the multi-agent part. In *independent Q-learning* (Tan 1993), each agent runs ordinary Q-learning on its own observations, actions and rewards, and treats the others as part of the environment. It scales to any number of agents, needs no communication and no knowledge of the others' actions, and works surprisingly often. It also fails in characteristic ways, which the cooperative matrix games of Claus and Boutilier (1998) make visible.

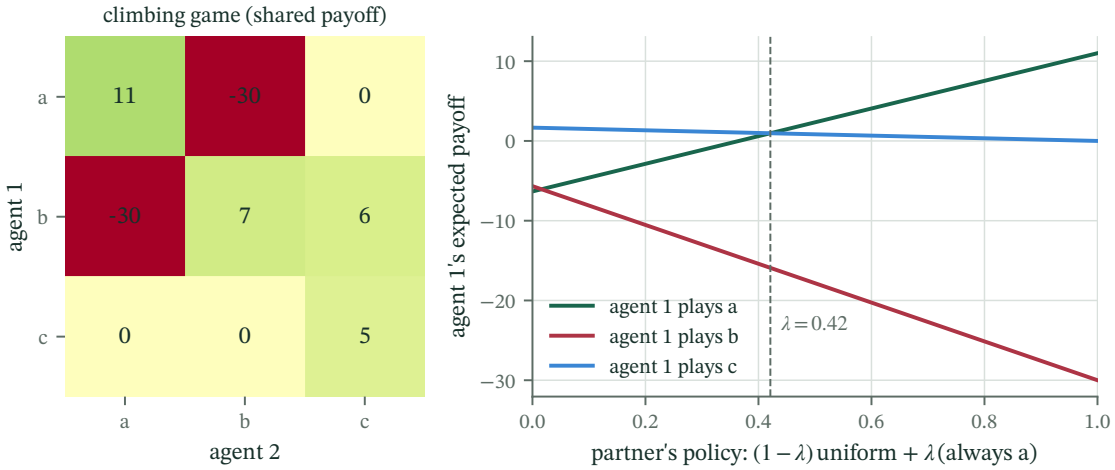


Figure 21.2. The climbing game of Claus and Boutilier (1998), in which both agents receive the same payoff (left), and the expected payoff of each of agent 1's actions when agent 2 plays a mixture of the uniform policy and the optimal action a (right). While agent 2 explores uniformly ($\lambda = 0$), the optimal action a looks worst; it looks best only once agent 2 already plays a with probability above 0.61 ($\lambda > 0.42$).

In the *climbing game* (Figure 21.2) two agents choose among a, b and c and share the payoff. The best joint action is (a, a), worth 11, but miscoordinating next to it is punished with -30. Suppose agent 2 is still exploring and plays uniformly at random. Agent 1's average payoffs are then $(11 - 30 + 0)/3 \approx -6.3$ for a, $(-30 + 7 + 6)/3 \approx -5.7$ for b and $(0 + 0 + 5)/3 \approx 1.7$ for c: the safe action c looks best and the optimal action a looks worst. Agent 1 learns to avoid a, agent 2 learns the same, and they settle on a poor but safe joint action. In 100 runs of two independent Q-learners (5,000 rounds each, exploration decaying from 1 to 0.05), the greedy joint action at

the end was (c, c), worth 5, in 92 runs and (b, c), worth 6, in 8, and never (a, a). This failure is called *relative overgeneralization* (Matignon et al. 2012): each agent averages its action's value over the partner's exploration, and the averages favor actions that are robust to a bad partner over actions that are excellent with a good one. In the right panel of Figure 21.2, the partner must already play a with probability above 0.61 before a becomes agent 1's best reply.

The *penalty game* shows a second failure, *miscoordination*. It has two optimal joint actions, (a, a) and (c, c), each worth 10, a safe middle action worth 2, and a penalty k for mixing up the two optima. With $k = -100$, independent learners settled on the safe (b, b) in all 100 of our runs: both optimal actions look terrible while the partner explores, and even agents that valued them could not agree on which optimum to choose. More generally, with several optimal equilibria, independent learners face an *equilibrium selection* problem that nobody tells them how to solve.

A third difficulty is the *non-stationarity* discussed in Section 21.1, which is especially damaging for deep RL agents that learn from replay: transitions stored while the partners behaved one way are replayed after they behave another way. Remedies include optimistic updates that ignore low rewards caused by a partner's exploration in deterministic games (*distributed Q-learning*, Lauer and Riedmiller (2000)), learners that are lenient early and strict later, and *joint-action learners*, which observe the partners' actions and learn values of joint actions (Claus and Boutilier 1998). The most widely used family, however, keeps the agents' policies separate but gives them a shared, centralized learning signal during training.

21.5 Centralized training with decentralized execution

In many cooperative tasks the agents must act on their own observations when deployed (robots with local sensors, units in a game that see only their surroundings), but during training, in a simulator, everything can be observed: the global state, all actions, all rewards. *Centralized training with decentralized execution* (CTDE) exploits this asymmetry. The policies are decentralized, $\pi_i(a_i \mid \tau_i)$ with τ_i agent i's own history of observations; the critics or value functions used to train them may use any information available in training. Solving decentralized POMDPs exactly is intractable (NEXP-complete even for two agents and a finite horizon, Bernstein et al. (2002)), so CTDE methods are approximate; their value lies in turning the multi-agent problem into something close to single-agent RL.

21.5.1 Value decomposition

A centralized value $Q_{\text{tot}}(s, a_1, \dots, a_n)$ of joint actions is easy to learn but hard to use: acting greedily requires a maximization over exponentially many joint actions, and it requires the global state. Value decomposition methods restrict the form of Q_{tot} so that each agent can act greedily on its own utility $Q_i(\tau_i, a_i)$, and the joint greedy action is still the greedy action of Q_{tot}. This requirement is called the *individual-global-max* (IGM) property (Son et al. 2019):

$$\arg\max_{a_1,\dots,a_n} Q_{\text{tot}}(s, a_1, \dots, a_n) \supseteq \Big(\arg\max_{a_1} Q_1(\tau_1, a_1), \dots, \arg\max_{a_n} Q_n(\tau_n, a_n)\Big).$$

Value decomposition networks (VDN, Sunehag et al. (2018)) use the sum, $Q_{\text{tot}} = \sum_i Q_i(\tau_i, a_i)$, trained end to end by Q-learning on the team reward: the TD error of Q_{tot} is backpropagated

into every agent network. IGM holds trivially, since a sum is maximized by maximizing each term. *QMIX* (Rashid et al. 2018) allows any *monotonic* combination: $Q_{\text{tot}} = f_s(Q_1, \dots, Q_n)$ with $\partial f_s / \partial Q_i \geq 0$, implemented by a mixing network whose weights are produced from the global state by *hypernetworks* and forced to be non-negative (Figure 21.3). Monotonicity is enough for IGM: increasing any one agent's utility never decreases Q_{tot}, so maximizing each utility maximizes the mixture. The state enters only through the mixing weights, so it is used in training and not needed for acting.

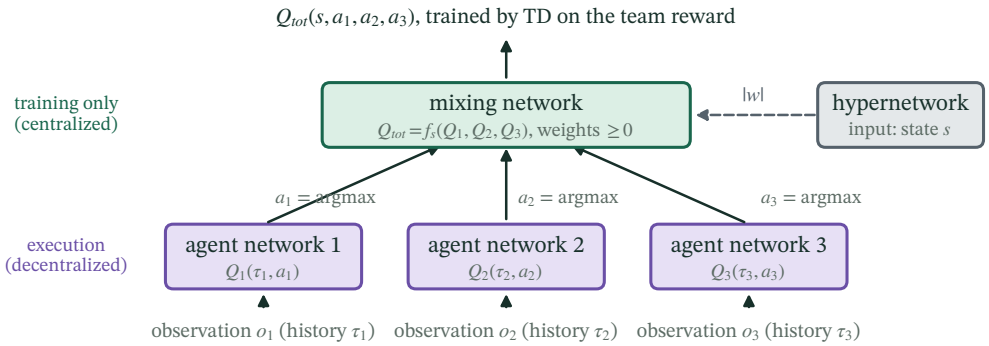


Figure 21.3. QMIX. Each agent network computes utilities $Q_i(\tau_i, a_i)$ from the agent's own observation history and acts greedily on them. During training, a mixing network combines the chosen utilities into Q_{tot}; its weights are generated from the global state s by a hypernetwork and made non-negative (for example by taking absolute values), which makes Q_{tot} monotonic in every Q_i.

The price is representational. A sum cannot represent interactions between agents, and a monotonic mixture cannot represent joint values in which the best action of one agent depends on what another does in a non-monotonic way. In the climbing game, the best additive fit to the payoffs (least squares with all joint actions weighted equally) is greedy at (c, c), the same mistake as the independent learners make (Exercise 21.11). Later methods relax the restriction: QTRAN (Son et al. 2019) learns a correction term that makes the factorization exact on the greedy action, and Weighted QMIX (Rashid et al. 2020) weights the projection toward the best joint actions. None removes the basic tension between expressiveness and easy decentralized maximization.

21.5.2 Centralized critics

Actor-critic methods (Algorithm 11.2) adapt to CTDE naturally: decentralized actors $\pi_i(a_i \mid \tau_i)$, trained with a critic that sees everything. *MADDPG* (Lowe et al. 2017) extends deterministic policy gradients (Theorem 13.1) by giving each agent a critic $Q_i(s, a_1, \dots, a_n)$ of the global state and all actions; because the critic conditions on everybody's actions, the transition law it models, $P(s' \mid s, a_1, \dots, a_n)$, does not change when the others' policies change, which removes the main source of non-stationarity from the critic's learning problem. It applies to cooperative, competitive and mixed settings, since each agent has its own critic and reward.

For teams, *COMA* (J. N. Foerster et al. 2018) addresses multi-agent credit assignment with a *counterfactual baseline*. With a centralized critic $Q(s, a)$ of the joint action $a = (a_i, a_{-i})$, agent i's advantage compares the value of the joint action with the average over agent i's own alternatives,

the others' actions held fixed:

$$A_i(s,a) = Q(s,a) - \sum_{a_i'} \pi_i(a_i' \mid \tau_i)\, Q\big(s,(a_i', a_{-i})\big).$$

It answers the question "how much better was my action than what I would typically have done, given what the others did?" The baseline does not depend on a_i, so by the argument of Theorem 11.2 it adds no bias to agent i's policy gradient (Exercise 21.12).

The most influential recent finding is that strong results do not require elaborate machinery. *MAPPO* (Yu et al. 2022) is PPO (Algorithm 12.2) with decentralized actors, parameters shared across agents, and a centralized value function; with careful implementation (value normalization, the right inputs to the critic, few epochs per batch) it was competitive with or better than the best off-policy methods on several standard cooperative benchmarks, with little tuning. Earlier, Schroeder de Witt et al. (2020) had reported that *independent* PPO, with no centralized critic at all, matched or beat state-of-the-art methods on many maps of the StarCraft Multi-Agent Challenge. Independent learning is a stronger baseline than its failure on matrix games suggests, partly because on-policy methods with small, clipped updates suffer less from the nonstationarity that plagues replay.

These comparisons rely on benchmarks. The *StarCraft Multi-Agent Challenge* (SMAC, Samvelyan et al. (2019)) asks teams of units, each controlled by its own policy with a limited field of view, to defeat built-in opponents. *SMACv2* (Ellis et al. 2023) was designed after its authors found that many original SMAC scenarios lacked enough randomness: policies that ignored their observations and acted only on the time step could win a substantial fraction of games. The *Hanabi* challenge (Bard et al. 2020) tests cooperation under imperfect information, where agents must communicate through their actions. Results on these benchmarks are sensitive to implementation details and to the evaluation protocol, so the advice of Section 23.4 applies with extra force.

PITFALL

A method that coordinates well with copies of itself may fail with any other partner. Agents trained together develop private conventions, such as an arbitrary choice between two equally good optima, that work only with the partners they were trained with. Evaluate cooperative agents with partners they have not trained with, as well as in self-play, and report both.

21.6 Imperfect-information games ●●

21.6.1 Extensive-form games and Kuhn poker

In poker a player does not know the opponent's cards, and the opponent's betting is informative precisely because it depends on them. Such games are described in *extensive form*: a tree whose nodes are *histories* (sequences of actions, including the moves of *chance*, such as dealing cards), each non-terminal history belonging to one player or to chance, with payoffs at the leaves. Hidden information is modeled with *information sets*: a player's information set is a set of histories that the player cannot tell apart when it is their turn, such as all deals in which they hold a given

card, followed by the same betting. A *behavioral strategy* σ_i assigns to each information set I of player i a distribution $\sigma_i(I)$ over the actions available there. We assume *perfect recall*: players never forget their own past actions or what they observed. For two-player zero-sum extensive-form games with perfect recall, an equilibrium can be computed by a linear program whose size is linear in the size of the game tree, the *sequence form* (Koller et al. 1996); the difficulty is that interesting games have astronomically large trees.

Kuhn poker (Kuhn 1950) is the smallest interesting example (Figure 21.4). The deck has three cards, Jack, Queen and King. Each player antes one chip and receives one card; the third card is not used. Player 1 may pass (check) or bet one chip. After a check, player 2 may check, which leads to a showdown for the antes, or bet; after any bet, the other player must fold, losing the ante, or call, which leads to a showdown for two chips each. Each player has six information sets (three cards times two decision points), each with two actions.

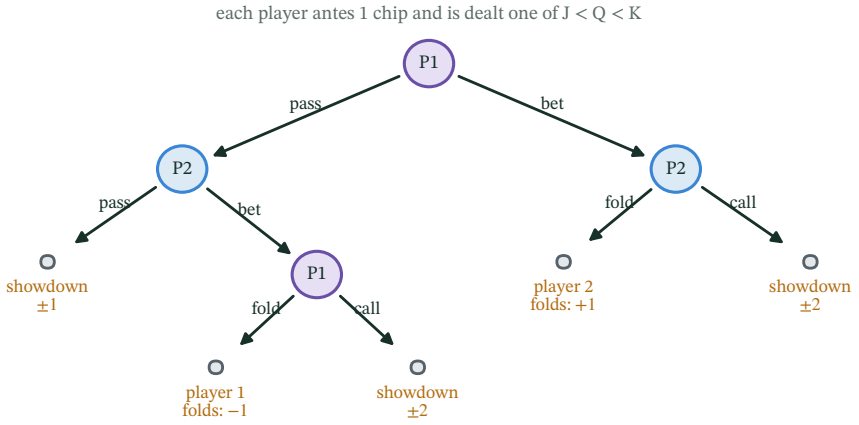


Figure 21.4. The betting tree of Kuhn poker, which follows the deal. Player 1 (P1) acts first; "pass" means check (or fold after a bet) and "bet" means bet (or call after a bet). The payoffs are those of player 1. Player 1's information set is its own card together with the betting so far: it cannot distinguish the deals in which player 2 holds either of the other two cards.

Kuhn solved the game by hand. In every equilibrium, player 1 checks with the Queen, bets the Jack with some probability $\alpha \in [0, 1/3]$ (a bluff), bets the King with probability 3α, and after a check and a bet calls with the King, folds the Jack, and calls with the Queen with probability $\alpha + 1/3$. Player 2 always bets or calls with the King; with the Queen it checks after a check and calls a bet with probability $1/3$; with the Jack it never calls but bluffs with probability $1/3$ after a check. Player 1's expected payoff is $-1/18$ per hand, whatever α: acting first is a disadvantage. Two features make this game different from an MDP. Good play is *randomized*, even with perfect knowledge of the rules: bluffing must be unpredictable. And the value of an information set depends on the *opponent's* strategy, because the opponent's betting determines what the cards behind it are likely to be; there is no value function of the player's observations alone that is independent of the strategies, and solving parts of the tree separately, as dynamic programming does, gives wrong answers. The tools of Chapter 19 (beliefs) and of this chapter (equilibria) are both needed.

21.6.2 Counterfactual regret minimization

Counterfactual regret minimization (CFR, Zinkevich et al. (2007)) computes approximate equilibria of two-player zero-sum extensive-form games by running a separate no-regret learner, regret matching, at every information set, with carefully weighted payoffs. It uses three kinds of quantities. For a strategy profile σ and a history h, the *reach probability* $\pi^\sigma(h)$ is the product of the probabilities of all actions on the way to h, chance's included; it factors into player i's own contribution $\pi_i^\sigma(h)$ and everybody else's, $\pi_{-i}^\sigma(h)$, chance included. The *counterfactual value* of information set I for player i is

$$v_i(\sigma, I) = \sum_{h \in I} \pi_{-i}^\sigma(h) \sum_{z \text{ terminal},\ h \sqsubseteq z} \pi^\sigma(h \to z)\, u_i(z), \tag{21.2}$$

where $\pi^\sigma(h \to z)$ is the probability of going from h to the leaf z and $u_i(z)$ is the payoff. In words: the payoff player i expects from I onward if it had played so as to reach I, with each history weighted by how likely chance and the opponent make it. Writing $v_i(\sigma, I, a)$ for the same quantity when i takes action a at I, the *counterfactual regret* of a at iteration t is $r^t(I, a) = v_i(\sigma^t, I, a) - v_i(\sigma^t, I)$.

Algorithm 21.2 Counterfactual regret minimization (CFR)

Input: a two-player zero-sum extensive-form game with perfect recall; a number of iterations T.

1. For every information set I and action a: cumulative regret $R(I, a) \leftarrow 0$ and strategy sum $S(I, a) \leftarrow 0$.
2. For $t = 1, \dots, T$:
   1. In every I, set $\sigma^t(I)$ by regret matching: $\sigma^t(I)(a) \propto \max(R(I, a), 0)$, or uniform if no regret is positive.
   2. Traverse the game tree once, computing for every I and a the counterfactual values $v_i(\sigma^t, I, a)$ and $v_i(\sigma^t, I)$ and the reach probability $\pi_i^{\sigma^t}(I)$ of the player i who acts at I.
   3. $R(I, a) \leftarrow R(I, a) + v_i(\sigma^t, I, a) - v_i(\sigma^t, I)$ for every I and a.
   4. $S(I, a) \leftarrow S(I, a) + \pi_i^{\sigma^t}(I)\, \sigma^t(I)(a)$ for every I and a.

Output: the average strategy $\bar{\sigma}^T(I)(a) = S(I, a) / \sum_b S(I, b)$.

The traversal in step 2 is a recursive walk over the tree that passes down the reach probabilities of both players and returns expected payoffs; for Kuhn poker it visits every history of all six deals once per iteration. The output is the *average* strategy, weighted by each player's own probability of reaching the information set; the current strategies σ^t do not converge in general, just as in Figure 21.1.

Theorem 21.5 Regret bound of CFR (Zinkevich et al. 2007)

Let Δ_i be the range of player i's payoffs, $|\mathcal{I}_i|$ the number of its information sets, and $|\mathcal{A}_i|$ the largest number of actions at any of them. Then (a) player i's average overall regret after T iterations is

at most the sum over its information sets of the positive parts of the average counterfactual regrets; and (b) with regret matching at every information set, it is at most $\Delta_i\,|\mathcal{I}_i|\sqrt{|\mathcal{A}_i|}/\sqrt{T}$. Consequently, in a two-player zero-sum game, the average strategy profile $\bar{\sigma}^T$ is a 2ϵ-equilibrium with $\epsilon = \max_i \Delta_i |\mathcal{I}_i|\sqrt{|\mathcal{A}_i|}/\sqrt{T}$.

Part (a) is the heart of CFR: the regret of the whole strategy decomposes into local regrets that can be minimized independently, which is why counterfactual values weight histories by the *opponent's* reach and not by the player's own. The proof is by induction over the tree (see Zinkevich et al. 2007); part (b) then follows from the regret bound of regret matching, and the last statement from Exercise 21.8.

The first iteration on Kuhn poker shows the mechanics. Both players start with uniform strategies. Take player 2's information set "King, facing a bet". It contains two histories, in which player 1 holds the Jack or the Queen; each has probability $\pi_{-2} = \frac{1}{6}\cdot\frac{1}{2} = \frac{1}{12}$ (the deal, and player 1's bet). Calling wins two chips in both, folding loses one. So $v(\text{call}) = 2\cdot\frac{1}{12}\cdot 2 = \frac{1}{3}$, $v(\text{fold}) = 2\cdot\frac{1}{12}\cdot(-1) = -\frac{1}{6}$, and under the uniform strategy $v = \frac{1}{12}$. The regrets are $+\frac{1}{4}$ for calling and $-\frac{1}{4}$ for folding, and from the second iteration on, regret matching calls with the King for sure. At player 1's information set "Queen, first to act", the same computation gives regrets $+\frac{1}{8}$ for betting and $-\frac{1}{8}$ for checking (Exercise 21.9), so the Queen bets in iteration 2, a mistake that later iterations correct.

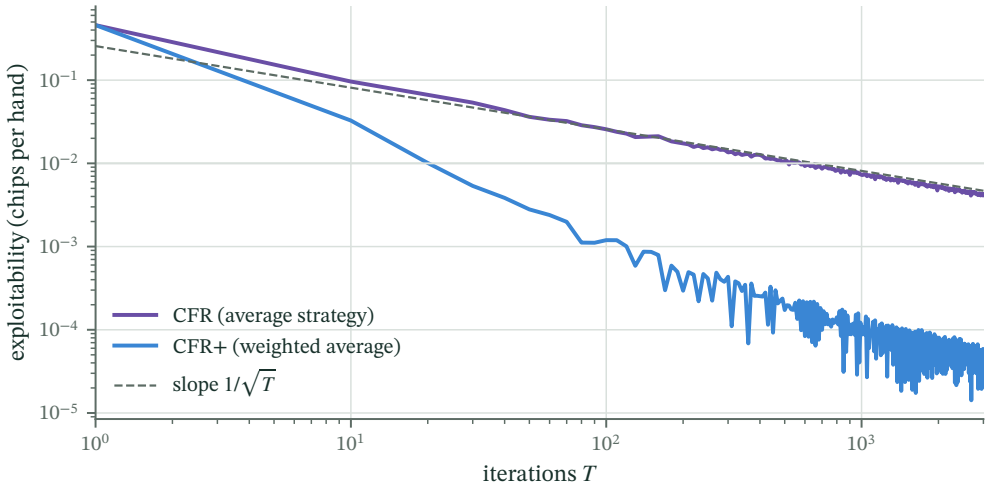


Figure 21.5. Exploitability of the average strategy on Kuhn poker for CFR and CFR+ (log–log scale; exploitability is half the sum of the two players' best-response gains, in chips per hand). CFR follows the $1/\sqrt{T}$ slope and falls below 0.01 (thin line) after about 500 iterations; CFR+ is much faster.

To measure progress we use the *exploitability* of a profile σ: the average, over the two players, of what each could gain by switching to a best response, $\frac{1}{2}\big[\max_{\sigma_1'} u_1(\sigma_1', \sigma_2) + \max_{\sigma_2'} u_2(\sigma_1, \sigma_2')\big]$, which is zero exactly at equilibria. In Kuhn poker a best response can be found by trying all $2^6 = 64$ pure strategies. Figure 21.5 shows that CFR's exploitability falls roughly like $1/\sqrt{T}$, from 0.458 for the uniform first strategy to 0.0073 after 1000 iterations, and the value of the average strategy is then within 0.001 of $-1/18$. The average strategy also lands in Kuhn's family: after

2000 iterations it bets the Jack with probability 0.19, the King with probability 0.61 (about three times as often) and almost never bets the Queen first.

Many refinements make CFR practical. *CFR+* (Tammelin 2014) floors the cumulative regrets at zero, alternates the updates of the two players and weights later iterations more in the average; on Kuhn poker it reaches an exploitability of about 0.0002 in 500 iterations, and it was used to essentially solve heads-up limit Texas hold'em (Bowling et al. 2015). *Monte Carlo CFR* (Lanctot et al. 2009) samples parts of the tree in each iteration instead of traversing all of it; *discounted CFR* (Brown and Sandholm 2019a) discounts early regrets; *Deep CFR* (Brown et al. 2019) replaces the tables by neural networks trained on sampled regrets.

21.6.3 Poker and beyond

Heads-up no-limit Texas hold'em has more than 10^{160} decision points, too many even for sampled CFR. The programs that beat professional players combined a coarse *blueprint* strategy, computed offline by a CFR variant on an abstracted game, with *search* during play. *Libratus* (Brown and Sandholm 2018) refined its blueprint by solving the subgame that was actually reached, with techniques that keep the refined strategy safe against exploitation, and in a 120,000-hand match in January 2017 it defeated four top professionals by a statistically significant margin. *DeepStack* (Moravčík et al. 2017) re-solved the game at every decision with a limited look-ahead, using a neural network that estimates counterfactual values at the depth limit, and was reported to beat professional players with statistical significance. *Pluribus* (Brown and Sandholm 2019b) extended the approach to six-player no-limit hold'em and was reported to be stronger than elite professionals, in experiments with one program against five professionals and with five copies of the program against one. With more than two players, equilibrium play no longer guarantees anything against coordinated or unexpected opponents; the program's success is an empirical result, not a theorem. *ReBeL* (Brown et al. 2020) brought the AlphaZero recipe of self-play, search and learned values to imperfect-information games by searching over *public belief states*, distributions over the private information given what all players have seen.

RL-based alternatives to CFR come from the self-play methods of Section 21.3.2. NFSP (Heinrich and Silver 2016) learns an approximate best response with deep Q-learning and the average policy by supervised learning, approximating fictitious play, and approached equilibrium in small poker variants; PSRO (Lanctot et al. 2017) grows a population of RL best responses and mixes them with a game-theoretic meta-solver. These methods need no game tree, only a simulator, which makes them applicable to video games and other domains in which CFR is impossible; the price is much weaker guarantees.

21.7 Theory ●●●

The complexity of equilibria. In two-player zero-sum games, equilibria are linear programs and therefore efficiently computable. In general-sum games they are not believed to be. Since a Nash equilibrium always exists, its computation is unlikely to be NP-hard (that would imply NP = co-NP, Exercise 21.15); Papadimitriou (1994) defined the class PPAD of search problems whose solutions are guaranteed to exist by a parity argument on a directed graph (the same argument

that proves Brouwer's and Sperner's theorems). Daskalakis et al. (2009) proved that computing a Nash equilibrium is PPAD-complete for games with three or more players, and Chen et al. (2009) extended the result to two players, even for approximate equilibria with an accuracy polynomial in the input size. Correlated equilibria, by contrast, are solutions of a linear program and can be computed efficiently. In Markov games the gap widens: Daskalakis et al. (2023) show that computing even approximate *stationary* coarse correlated equilibria of general-sum discounted Markov games is PPAD-hard, while non-stationary ones can be computed efficiently in finite-horizon games. If equilibria are hard to compute, we should not expect simple learning dynamics to find them.

No-regret learning. Learning theory offers a different guarantee. If every player of a general-sum game uses an algorithm with vanishing external regret, the empirical distribution of their joint play converges to the set of coarse correlated equilibria (Hannan 1957); with vanishing *swap* (or internal) regret, it converges to the set of correlated equilibria (Foster and Vohra 1997; Hart and Mas-Colell 2000). In two-player zero-sum games, the average strategies converge to the set of Nash equilibria, which is how CFR works. Nash equilibria themselves are out of reach of such dynamics in general: Hart and Mas-Colell (2003) show that no *uncoupled* dynamics, in which each player's update depends only on its own payoffs, converge to Nash equilibria in all games.

Last iterates. Averages converge; the strategies themselves often do not. In zero-sum games, multiplicative weights moves away from interior equilibria (Bailey and Piliouras 2018), and continuous-time follow-the-regularized-leader dynamics are *Poincaré recurrent*, returning arbitrarily close to their starting point forever (Mertikopoulos et al. 2018), as in Figure 21.1. *Optimistic* updates, which use the last gradient as a prediction of the next, converge in the last iterate in bilinear zero-sum games (Daskalakis et al. 2018). For zero-sum Markov games, Daskalakis et al. (2020) proved that independent policy gradient methods, with suitably separated step sizes for the two players, converge to a min-max equilibrium; Wei et al. (2021) proved last-iterate convergence for decentralized optimistic gradient descent/ascent; and Cen et al. (2021) showed that entropy regularization with extragradient updates gives linear convergence to the regularized (quantal response) equilibrium. In cooperative and *potential* games, where all players' incentives are captured by one function, independent policy gradient converges to Nash equilibria (Leonardos et al. 2022). On the statistical side, Bai and Jin (2020) gave self-play algorithms for zero-sum Markov games with regret guarantees of order $\sqrt{T}$.

Learning about the other learners. All of these methods treat the opponent's learning as noise. *Learning with opponent-learning awareness* (LOLA, J. Foerster et al. (2018)) instead differentiates through it: agent 1 updates its parameters θ_1 to maximize $V^1(\theta_1, \theta_2 + \Delta\theta_2)$, where $\Delta\theta_2 = \eta \nabla_{\theta_2} V^2(\theta_1, \theta_2)$ is the opponent's anticipated gradient step. To first order this adds a term $\eta\,(\nabla_{\theta_2} V^1)^{\top} \nabla_{\theta_1} \nabla_{\theta_2} V^2$ to the naive gradient, which accounts for how agent 1's policy changes the direction in which agent 2 learns. In the iterated prisoner's dilemma, LOLA agents learned reciprocating, tit-for-tat-like strategies and cooperated, where naive learners defected. LOLA can also fail to preserve stable equilibria; *stable opponent shaping* (Letcher et al. 2019) modifies it to keep LOLA's benefits while provably preserving them.

RESEARCH NOTE

Open questions: Which natural learning dynamics converge in the last iterate in general-sum Markov games, and to what? Can sample-efficient learning of correlated or coarse correlated equilibria scale to many agents without an exponential dependence on their number? How should a learner reason about opponents who are themselves learning, without assuming they use a particular algorithm?

21.8 Landmarks and frontiers ●●●

A few systems mark what multi-agent RL has achieved at scale; the claims below are as reported by their authors.

AlphaStar (Vinyals et al. 2019) played the real-time strategy game StarCraft II, which combines imperfect information, long horizons and a huge action space. It was first trained by imitation of human games and then by multi-agent RL in a *league*: main agents trained against a mixture of all past league members (a prioritized form of fictitious self-play), while dedicated *exploiter* agents searched for weaknesses of the main agents and of the league as a whole. It was rated at Grandmaster level for all three StarCraft races, above 99.8% of officially ranked human players. *OpenAI Five* (Berner et al. 2019) played the five-on-five team game Dota 2 with five copies of a recurrent policy trained by PPO in self-play for about ten months, at very large scale; in April 2019 it defeated the reigning world champion team. *DeepNash* (Perolat et al. 2022) learned Stratego, a board game with hidden pieces and a game tree far too large for CFR, from self-play with a model-free method based on regularized Nash dynamics (Perolat et al. 2021), which converges in the last iterate; it reached a top-three all-time ranking on an online platform for expert human players.

CICERO (Meta Fundamental AI Research Diplomacy Team (FAIR) et al. 2022) played Diplomacy, a seven-player game of alliances and betrayal in which players negotiate in natural language. It combined a language model that generates messages conditioned on intended actions with a planning module that predicts the other players' moves, using search regularized toward a model of human play so that its behavior remained understandable to people (Jacob et al. 2022). In 40 games of an anonymous online league it achieved more than double the average score of the human players and ranked in the top 10% of participants who played more than one game. Unlike two-player zero-sum games, Diplomacy rewards cooperation with humans who follow human conventions, and pure self-play produced agents that did not.

Language-model agents now raise multi-agent questions of their own. Agents that negotiate can drift away from human language when trained against each other with RL (Lewis et al. 2017); populations of language agents simulate social behavior (Park et al. 2023); and several proposals use interaction between models to improve them, as in *debate*, where models argue for opposing answers before a judge (Irving et al. 2018), or multi-agent discussion to improve factual accuracy and reasoning (Du et al. 2024). When language models are trained by RL against each other, or interact with people who adapt to them, the questions of this chapter (equilibrium, exploitability, conventions, non-stationarity) return in a new setting; Chapter 22 discusses RL for language models.

RESEARCH NOTE

Open questions: How should agents be trained to cooperate with humans and with agents they have never met (zero-shot coordination)? How can multi-agent systems built from language models be evaluated when their behavior depends on who else is present? What does equilibrium analysis say about AI systems that train against each other at scale, and how can such training be kept from finding exploits that are useless or harmful outside the training population?

21.9 Summary

- Other learning agents make an agent's environment non-stationary and its objective dependent on what the others do; settings are cooperative, competitive or mixed.
- A Nash equilibrium is a profile of mutual best responses; it exists in every finite game but may be mixed, may not be unique and may be inefficient (the prisoner's dilemma, chicken). Correlated equilibria, defined by a joint recommendation, are found by linear programming and can be better for everyone.
- In two-player zero-sum games the minimax theorem gives a value and safe strategies, computable by linear programming. In zero-sum Markov games, Shapley's operator is a γ-contraction, and optimal policies are stochastic in general.
- Minimax-Q learns zero-sum Markov games from experience, replacing the max of Q-learning by the value of a matrix game; on the striker-and-goalkeeper game it learns the exact values to within about 0.01, while a Q-learner trained against a random keeper is exploitable.
- Self-play is powerful but can cycle; gradient play spirals away from the equilibrium of matching pennies. No-regret learners (regret matching, fictitious play in zero-sum games) have averages that converge, which is why the deployed policy is often the average.
- Independent learners are simple and often strong, but they fail through non-stationarity, miscoordination and relative overgeneralization, as in the climbing game.
- Centralized training with decentralized execution uses global information only during training: value decomposition (VDN, QMIX) with the individual-global-max property, centralized critics (MADDPG, COMA's counterfactual baseline), and MAPPO, a strong and simple baseline.
- (●●) In imperfect-information games, counterfactual regret minimization runs regret matching at every information set; on Kuhn poker its average strategy converges to the equilibrium of value $-1/18$. CFR with search beat professionals at heads-up and six-player poker.
- (●●●) Nash equilibria are PPAD-hard to compute in general; no-regret dynamics reach coarse correlated equilibria; last-iterate convergence needs special methods. The landmark systems (AlphaStar, OpenAI Five, DeepNash, CICERO) combine self-play, populations and, increasingly, language.

The next chapter turns to the most widespread current use of reinforcement learning: training language models.

21.10 Further Reading

For game theory, Shoham and Leyton-Brown (2009) is a thorough introduction written for computer scientists, and Nisan et al. (2007) covers the algorithmic side, including the complexity of equilibria. Fudenberg and Levine (1998) is the classic treatment of learning in games, and Cesa-Bianchi and Lugosi (2006) the standard reference on regret and no-regret learning. The book of Albrecht et al. (2024) is a modern textbook on multi-agent RL, and K. Zhang et al. (2021) a careful survey of its theory.

The founding papers are short and readable: Shapley (1953) for stochastic games, Littman (1994) for Markov games in RL and minimax-Q, and Claus and Boutilier (1998) for independent and joint-action learners in cooperative games, including the climbing and penalty games. Bowling and Veloso (2002) discuss what convergence in multi-agent learning should mean and introduce WoLF. For coordination failures of independent learners, see the survey of Matignon et al. (2012).

For deep cooperative MARL, read Rashid et al. (2018) (QMIX), J. N. Foerster et al. (2018) (COMA) and Lowe et al. (2017) (MADDPG), then Yu et al. (2022) and Schroeder de Witt et al. (2020) for the strength of PPO-based methods and Ellis et al. (2023) for the benchmark caveats.

For imperfect-information games, Zinkevich et al. (2007) introduced CFR; Brown and Sandholm (2018), Moravčík et al. (2017) and Brown and Sandholm (2019b) describe the poker systems; Lanctot et al. (2017) presents PSRO.

(●●●) For theory, Daskalakis et al. (2009) and Chen et al. (2009) on the complexity of Nash equilibria, Hart and Mas-Colell (2003) on why uncoupled dynamics cannot always find them, Daskalakis et al. (2020) and Wei et al. (2021) on policy gradient methods in zero-sum Markov games, and J. Foerster et al. (2018) on learning with opponent awareness. The landmark papers Vinyals et al. (2019), Berner et al. (2019), Perolat et al. (2022) and Meta Fundamental AI Research Diplomacy Team (FAIR) et al. (2022) each contain a wealth of engineering detail.

21.11 Exercises

Exercise 21.1 ● Equilibria of two-by-two games

Find all Nash equilibria, pure and mixed, of (a) the prisoner's dilemma, (b) chicken and (c) the stag hunt with row payoffs $A = \left(\begin{smallmatrix} 4 & 0 \\ 3 & 3 \end{smallmatrix}\right)$ (actions stag, hare) and $B = A^{\mathsf{T}}$. For each equilibrium give both players' expected payoffs. (d) In the stag hunt, which equilibrium would you expect players to choose, and why is the question not answered by the definition of equilibrium?

Exercise 21.2 ● Minimax by linear programming

The row player's payoffs in a zero-sum game are $A = \left(\begin{smallmatrix} 2 & -1 & 1 \\ -1 & 1 & 0 \end{smallmatrix}\right)$. (a) Write the linear program (21.1) for the row player. (b) Plot the row player's payoff against each column as a function of $p = x_1$ and find the maximin strategy and the value graphically. (c) Find the column player's minimax strategy. Which column does it never use, and why?

Exercise 21.3 • Best-response dynamics

In *best-response dynamics*, the players take turns switching to a best response to the other's current pure action. (a) Show that in matching pennies the dynamics cycle through all four pure profiles. (b) Show that in the stag hunt they reach an equilibrium from every starting profile, and say which one is reached from where. (c) What happens in chicken if both players start by daring?

Exercise 21.4 • Fictitious play by hand

Run fictitious play on rock-paper-scissors for both players, starting from one round in which both played rock, and breaking ties in the order rock, paper, scissors. (a) Write down the actions of the next four rounds and the empirical frequencies after each. (b) Explain why the actions come in runs, and why the runs get longer. (c) Why do the empirical frequencies still converge to $(1/3, 1/3, 1/3)$?

Exercise 21.5 • Regret matching by hand

A row player uses regret matching in rock-paper-scissors. In round 1 it plays rock and the opponent plays paper; in round 2 it plays scissors and the opponent plays rock. (a) Compute the cumulative regret of each action after rounds 1 and 2. (b) Give the regret-matching strategies for rounds 2 and 3. (c) Why does regret matching never put weight on an action with negative cumulative regret, and what does it do when all regrets are negative?

Exercise 21.6 • The striker improves

In the striker-and-goalkeeper game of Example 21.3, suppose the striker's close-range shot against a keeper who stays improves from 30% to 40%. (a) Recompute the value and both players' strategies at close range. (b) Propagate the change to the edge of the box and to far away. (c) The striker became better at shooting from close range. Does he shoot more or less often there? Does the keeper stay more or less often? Explain the logic of the indifference principle behind your answers.

Exercise 21.7 •• Shapley's operator is a contraction

(a) Prove that for matrices A and B of the same size, $|\operatorname{val}(A) - \operatorname{val}(B)| \le \max_{ij} |A_{ij} - B_{ij}|$. (b) Deduce that the Shapley operator is a γ-contraction in the maximum norm. (c) Show by an example that val is not linear: find A and B with $\operatorname{val}(A + B) \neq \operatorname{val}(A) + \operatorname{val}(B)$. Why does Shapley's value iteration converge nevertheless?

Exercise 21.8 •• From no regret to equilibrium

In a zero-sum matrix game with payoff matrix A, the players play x^t and y^t for $t = 1, \dots, T$, with average regrets R_1^T and R_2^T (for the column player, regret is measured on its own payoff $-x^{\top} A y$).

Let $\bar{x}$ and $\bar{y}$ be the average strategies. (a) Show that $\max_x x^\top A\bar{y} - \min_y \bar{x}^\top A y = R_1^T + R_2^T$. (b) Deduce that neither player can gain more than $R_1^T + R_2^T$ by deviating from $(\bar{x}, \bar{y})$, and that $\bar{x}^\top A\bar{y}$ is within $R_1^T + R_2^T$ of the value. (c) Why does the argument fail for general-sum games, and what does the average of no-regret play converge to there?

Exercise 21.9 ●● Counterfactual regrets by hand

At the first iteration of CFR on Kuhn poker both players use uniform strategies. (a) Verify the counterfactual values and regrets of player 2's information set "King, facing a bet" given in Section 21.6.2. (b) Compute the counterfactual values of betting and checking at player 1's information set "Queen, first to act", and show that the regrets are $+1/8$ and $-1/8$. (c) Explain why counterfactual values weight histories by the opponent's and chance's reach probabilities but not by the acting player's own.

Exercise 21.10 ●● Indifference in Kuhn poker

(a) Suppose player 2 follows the equilibrium strategy of Section 21.6.1. Show that player 1, holding the Queen, after checking and facing a bet, is indifferent between calling and folding. (b) Suppose player 1 bets the Jack with probability α and the King with probability 3α, with $\alpha > 0$. Show that player 2, holding the Queen and facing a bet, is indifferent between calling and folding. (c) Explain why both indifferences are needed for an equilibrium, and why the bluffing frequencies are tied to the betting frequency with the King.

Exercise 21.11 ●● What value decomposition can represent

(a) Fit VDN to the climbing game by least squares with all nine joint actions weighted equally: show that $Q_1(a) + Q_2(b)$ equals the row mean of a plus the column mean of b minus the overall mean, and find the greedy joint action. (b) Show that no function $f(q_1, q_2)$ that is non-decreasing in each argument can represent the coordination game $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ exactly as $f(Q_1(a), Q_2(b))$. (c) Show that any monotonic mixing satisfies the individual-global-max property, so that QMIX's decentralized greedy actions are greedy for Q_{tot}.

Exercise 21.12 ●● The counterfactual baseline

(a) Show that COMA's baseline $b(s, a_{-i}) = \sum_{a_i'} \pi_i(a_i' \mid \tau_i)\, Q(s, (a_i', a_{-i}))$ adds no bias to agent i's policy gradient, even though it depends on the other agents' actions. (b) Two agents share the reward 1 if both choose action 1 and 0 otherwise; agent 1 currently chooses each action with probability $1/2$, and agent 2 chooses 1 with probability 0.9. With a correct critic, compute agent 1's COMA advantages of its two actions when agent 2 chose 1 and when it chose 0. (c) Compare with the advantage computed with a baseline that does not depend on agent 2's action, and explain what the counterfactual baseline gains.

Exercise 21.13 ● Relative overgeneralization

(a) Verify the expected payoffs of agent 1's three actions in the climbing game when agent 2 plays uniformly, and when agent 2 plays $(1-\lambda)$ uniform plus λ "always a". (b) Find the smallest λ for which a is agent 1's best reply, and the corresponding probability that agent 2 plays a. (c) Explain why independent learners in the penalty game with $k=-100$ settle on the safe action, and why the problem disappears as $k \to 0$.

Exercise 21.14 ● Programming: self-play in rock-paper-scissors

Extend Lab 21.1. (a) Implement *naive self-play*, in which each player best-responds to the other's action in the previous round, and fictitious play. (b) For both, plot over 3000 rounds the exploitability of the current strategies and of the average strategies. (c) Explain the results. Which strategy would you deploy against an unknown opponent?

Exercise 21.15 ●●● Why is Nash hard?

(a) Explain the class PPAD and the "end of the line" problem in your own words, and why a problem with a guaranteed solution cannot be NP-hard unless NP = co-NP. (b) Sketch why the existence proof of Nash equilibria (via Brouwer's fixed-point theorem or Sperner's lemma) does not give an efficient algorithm. (c) Contrast with zero-sum games and correlated equilibria, which are computable by linear programming. (d) Discuss what PPAD-hardness implies for multi-agent RL algorithms that claim to find Nash equilibria of general-sum games, citing at least two of the results in Section 21.7.

Exercise 21.16 ●●● Equilibrium selection

Many games, from the stag hunt to traffic conventions, have several equilibria. (a) Explain *payoff dominance* and *risk dominance* in the stag hunt and compute which equilibrium each selects. (b) Summarize one theory of how learning dynamics select equilibria in the long run (for example stochastic stability). (c) In cooperative MARL, agents trained together may settle on arbitrary conventions that fail with new partners. Describe one approach to *zero-shot coordination* and its limits. (d) Propose an experiment, using the code of Lab 21.1, that tests which equilibrium of the stag hunt independent Q-learners select and how this depends on the payoffs.

Part VII

Reinforcement Learning in the World

The last part looks outward. Chapter 22 shows how reinforcement learning trains and aligns large language models. Chapter 23 collects the craft of making reinforcement learning work: implementation, debugging, honest evaluation and safety. Chapter 24 surveys the open problems and offers advice for readers who want to work on them.

CHAPTER 22

Reinforcement Learning for Language Models

WHAT YOU WILL LEARN

- Describe a language model as a policy, both as a contextual bandit that chooses whole responses and as a token-level Markov decision process, and explain what reinforcement learning adds to supervised fine-tuning.
- Explain the three stages of reinforcement learning from human feedback, and fit a Bradley–Terry reward model to preference data.
- Derive the optimal policy of the KL-regularized objective and use it to explain what the KL strength β controls.
- Derive direct preference optimization from that optimum, and state what it guarantees and what it does not (●●).
- Implement GRPO and the leave-one-out baseline for verifiable rewards, and explain the biases of GRPO's normalizations.
- Compare the estimators k_1, k_2 and k_3 of the KL divergence, and the gradients they produce in a loss and in a reward (●●).
- Recognize reward hacking and overoptimization, and discuss the open problems: exploration in reasoning, what reinforcement learning adds to a pretrained model, process rewards and scalable oversight (●●●).

A pretrained language model is an excellent predictor of text, and predicting text is not the same as being useful. Ask a pretrained model a question and it may answer it, or continue with three more questions, or invent a plausible citation, because all of these are likely continuations of text found on the internet. The assistants that people use every day are made from pretrained models by a second phase of training, called *post-training*, and since 2022 reinforcement learning has been a central part of it.

Two kinds of reward drive this work. The first is human judgment: people compare two responses and say which is better, a model of their preferences becomes the reward, and the language model is trained to earn it. This is *reinforcement learning from human feedback* (RLHF). The second is automatic checking: a mathematics answer can be compared with the known answer, and a program can be run against tests. Training against such checks, called *reinforcement learning with verifiable rewards*, has since 2024 produced "reasoning models" that write long chains of thought before they answer.

The setting is unusual for reinforcement learning, and its peculiarities shape the algorithms. Ac-

tions are tokens from a vocabulary of tens of thousands; transitions are known and deterministic; an episode is a response that can run to thousands of steps; the reward usually arrives once, at the end; and the starting policy is already very capable, so the job is to adjust it rather than to learn from scratch. Much of the book meets here: policy gradients (Chapter 11) and PPO (Chapter 12), entropy regularization (Chapter 13), learning rewards from preferences (Chapter 18), learning from fixed data (Chapter 17) and search (Chapter 8).

Section 22.1 casts a language model as a policy, and Section 22.2 describes the RLHF pipeline. Section 22.3 finds the best policy of the KL-regularized objective in closed form, which explains most of what follows: PPO for language models (Section 22.4), direct preference optimization (Section 22.5), and the critic-free methods for verifiable rewards (Section 22.6). Section 22.7 (●●) studies how the KL divergence is estimated from samples. Section 22.8 is about what goes wrong when a proxy reward is optimized too hard. Section 22.9 (●●) treats rewards for intermediate steps and search at test time, and Section 22.10 (●●●) surveys open problems. On a first reading, take Section 22.1 to 22.6 and Section 22.8.

22.1 Language models as policies

To apply reinforcement learning we must say what the agent, the actions, the states and the rewards are. For a language model there are two natural answers, and both are useful.

The model. A language model works with *tokens*, pieces of words from a fixed *vocabulary* $\mathcal{V}$ that typically holds 30,000 to 200,000 entries. A *prompt* x is a sequence of tokens. The model writes a *response* $y = (y_1, \dots, y_L)$ one token at a time, drawing each token from a distribution that depends on the prompt and on the tokens written so far:

$$\pi_\theta(y \mid x) = \prod_{t=1}^{L} \pi_\theta(y_t \mid x, y_{<t}).$$

Here $y_{<t} = (y_1, \dots, y_{t-1})$ is the *prefix* before token t, θ are the parameters of the network (a transformer, in practice), and the length L is itself random: generation stops when the model emits a special end-of-sequence token or reaches a maximum length. In words: the probability of a response is the product of the probabilities of its tokens, so its logarithm is a sum, $\log \pi_\theta(y \mid x) = \sum_t \log \pi_\theta(y_t \mid x, y_{<t})$. We write $\mathcal{Y}$ for the set of all possible responses.

The response-level view: a contextual bandit. Treat the whole response as one action. A prompt x arrives from a distribution $\mathcal{D}$ of prompts, the model chooses a response $y \sim \pi_\theta(\cdot \mid x)$, and a reward $r(x, y)$ grades it. Nothing else happens: there is no next state. This is a *contextual bandit* (Section 2.10) whose context is the prompt, and the objective is

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}} \, \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \big[r(x, y) \big].$$

The action set is astronomically large, about $|\mathcal{V}|^L$ responses, but the policy is easy to sample from and its log-probability is easy to differentiate, which is all a policy-gradient method needs ((11.4)).

The token-level view: a Markov decision process. Treat each token as an action. The state at step t is the prompt together with the prefix, $s_t = (x, y_{<t})$; the action is the next token, $a_t = y_t$;

the transition appends it, $s_{t+1} = (x, y_{\le t})$, deterministically; and the episode ends at the end-of-sequence token. The reward is zero for every token except the last, which receives $r(x, y)$, and there is no discounting ($\gamma = 1$), because every episode is finite. This MDP is peculiar in four ways. Its dynamics are known and deterministic, so planning and search cost only computation (Section 22.9). Its state space is a tree: every state is reached by exactly one history, so the state *is* the history. Its horizon is long, often thousands of tokens, with a single reward at the end, so credit assignment is hard. And its starting policy is a strong pretrained model rather than a random one.

The two views describe the same thing. The bandit view is the natural language for objectives and optimal policies (Section 22.3 and 22.5); the token view is needed by algorithms that assign credit to individual tokens, such as PPO with a value function (Section 22.4) and process rewards (Section 22.9).

Example 22.1 A language model small enough to enumerate

The labs use a toy task with a vocabulary of four tokens, 1, 2, 3 and the end token eos. The prompt names a target, for instance "make 4", and a response is correct if its digits add up to the target and it ends with eos; responses are at most six tokens long. A program checks the answer, so the reward is 1 for a correct response and 0 otherwise.

Suppose the model is uniform: every token has probability $\frac{1}{4}$ after every prefix. Then the response 1 3 eos has probability $(\frac{1}{4})^3 = \frac{1}{64} \approx 0.016$. The target 4 has seven correct responses: three with two digits (1 3, 3 1, 2 2), three with three digits (1 1 2 in some order) and one with four (1 1 1 1). Their total probability is

$$3 \cdot \left(\tfrac{1}{4}\right)^3 + 3 \cdot \left(\tfrac{1}{4}\right)^4 + \left(\tfrac{1}{4}\right)^5 = \tfrac{48+12+1}{1024} \approx 0.060.$$

In all there are 1,093 possible responses, few enough that the labs compute every probability and expectation exactly. As an MDP, the task has a state for every prefix (364 of them) and a reward only at the end.

♦

22.1.1 Why reinforcement learning?

A simpler way to adapt a language model is *supervised fine-tuning* (SFT): collect good responses written by people and maximize their log-likelihood. This is behavior cloning (Algorithm 18.1). Reinforcement learning earns its extra complexity for four reasons.

Judging is easier than demonstrating. Most people cannot write a perfect summary of a legal contract or a proof of a hard identity, but many can tell which of two attempts is better. A reward learned from such comparisons can push the model beyond the quality of the demonstrations it was given.

Some rewards cannot be differentiated. Whether a program passes its unit tests, or whether a final answer equals the known answer, is a yes-or-no fact about a whole response. Gradient descent cannot pass through it, but a policy gradient needs only samples and their rewards.

The model should learn from its own outputs. In supervised training every prefix comes from the demonstrator ("teacher forcing"); at test time the model conditions on its own earlier tokens, whose distribution is different, and a small early mistake can lead it into prefixes it never saw in training. This mismatch is called *exposure bias* (Bengio et al. 2015; Ranzato et al. 2016), and it is the compounding-error problem of behavior cloning (Theorem 18.1). Reinforcement learning trains on-policy, on the model's own samples, so it sees and learns from the consequences of its own mistakes.

Many responses are correct. Supervised learning pulls the model toward one particular reference response; a reward accepts every response that does the job.

PITFALL

In this chapter "the reward" is rarely the goal itself. A reward model is a learned and imperfect imitation of human judgment, and a checker is a program with gaps: a unit test does not test everything. A policy optimized hard enough finds the places where the reward and the intention differ. Section 22.8 is devoted to this.

CHECKPOINT

1. In the token-level MDP, what is the state after the model has written three tokens of its response?
2. Why is the discount factor 1 in this MDP?
3. In Example 22.1, what is the probability of the correct response 2 2 eos under the uniform model?

Answers.

1. The prompt together with the three tokens: $s_3 = (x, y_1, y_2, y_3)$.
2. Every response ends after a bounded number of tokens, so the return is a finite sum without discounting; and the reward arrives only at the end, so discounting would merely shrink it by a factor that depends on the length.
3. $(\frac{1}{4})^3 = \frac{1}{64}$: three tokens, each with probability $\frac{1}{4}$.

22.2 The RLHF pipeline

The recipe that made reinforcement learning a standard step in training assistants is shown in Figure 22.1. It was developed on simple control tasks (Christiano et al. 2017), applied to text continuation and summarization (Ziegler et al. 2019; Stiennon et al. 2020), and scaled up in InstructGPT (Ouyang et al. 2022). It has three stages.

Stage 1: supervised fine-tuning. People write demonstrations (x, y) of the desired behavior, and the pretrained model is fine-tuned to maximize $\sum \log \pi_\theta(y \mid x)$ on them. The result, π_{SFT}, has learned the format and style of a helpful answer. It becomes the starting point of stage 3 and the reference model π_{ref} that stage 3 must stay close to.

Stage 2: reward modeling. For many prompts, the SFT model writes several responses and people say which they prefer. The data are triples (x, y_w, y_l), where y_w ("winner") was preferred to

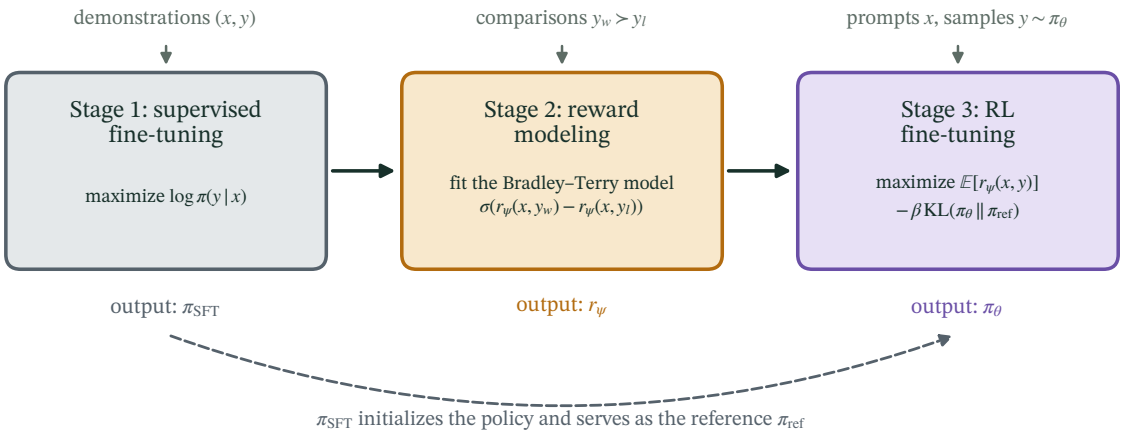


Figure 22.1. The three stages of reinforcement learning from human feedback. Stage 1 imitates demonstrations, stage 2 learns a reward from comparisons, and stage 3 optimizes the policy against that reward while a KL penalty keeps it close to the stage-1 model. Each stage's output feeds the next.

y_l ("loser"). The *Bradley–Terry model* (Bradley and Terry 1952) connects preferences to a scalar reward ((18.5)):

$$\Pr(y_w \succ y_l \mid x) = \sigma\big(r(x, y_w) - r(x, y_l)\big), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

In words: the better the reward of y_w compared with that of y_l, the more likely it is to be preferred, and equal rewards mean a coin flip. The *reward model* r_ψ with parameters ψ is usually the SFT network itself, with its output layer replaced by one that reads off a single number at the last token. It is fitted by maximum likelihood, that is, by minimizing

$$\mathcal{L}(\psi) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}}\big[\log \sigma\big(r_\psi(x, y_w) - r_\psi(x, y_l)\big)\big],$$

a logistic regression on the difference of two rewards. InstructGPT asked labelers to rank between four and nine responses at a time and used every pair from each ranking (Ouyang et al. 2022). Section 18.7 treats learning rewards from preferences in general.

For a feel for the numbers, suppose the reward model currently gives $r_\psi(x, y_w) = 1.2$ and $r_\psi(x, y_l) = 0.4$. It predicts the observed preference with probability $\sigma(0.8) = 0.690$, the loss on this pair is $-\log 0.690 = 0.371$, and the derivative of the loss with respect to the margin $r_\psi(x, y_w) - r_\psi(x, y_l)$ is $-(1 - 0.690) = -0.310$: gradient descent widens the margin, and it pushes less as the prediction becomes confident.

The Bradley–Terry probabilities depend only on *differences* of rewards for the same prompt. Adding any function $c(x)$ of the prompt alone to the reward leaves every preference probability unchanged, so preference data determine the reward only up to such a shift. The scale is fixed by the data: if labelers are noisy, the fitted rewards are close together. The shift does not change the policy of stage 3, and the scale acts only through the ratio of the reward to β, so rescaling the reward is the same as changing β (Exercise 22.3). Both facts mean that a reward model's raw numbers have no meaning on their own, and implementations usually normalize them.

Stage 3: reinforcement-learning fine-tuning. The policy π_θ, initialized to $\pi_{\text{ref}} = \pi_{\text{SFT}}$, is trained to maximize the reward model's score while staying close to the reference:

$$\max_\theta \mathbb{E}_{x\sim\mathcal{D}}\Big[\,\mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\big[r_\psi(x,y)\big] - \beta\, D_{\text{KL}}\big(\pi_\theta(\cdot\mid x)\,\|\,\pi_{\text{ref}}(\cdot\mid x)\big)\Big], \tag{22.1}$$

where $D_{\text{KL}}(p\,\|\,q) = \sum_y p(y)\log\frac{p(y)}{q(y)}$ is the Kullback–Leibler divergence between the response distributions and $\beta > 0$ sets the strength of the penalty. In words: earn a high reward, but pay β for every nat of divergence from the reference model. The penalty does three jobs. It keeps the policy where the reward model is trustworthy, since the reward model was trained on responses from π_{SFT} and is less accurate on very different ones. It preserves the fluency and diversity of the reference model. And it keeps the policy from collapsing onto one high-scoring response. The next section shows exactly what (22.1) asks for, and Section 22.4 shows how it is optimized in practice.

What each stage contributes. In InstructGPT, labelers preferred the outputs of a 1.3-billion-parameter model trained with this pipeline to those of the 175-billion-parameter GPT-3, a model over 100 times larger (Ouyang et al. 2022). The same study reported gains in truthfulness and less toxic output, with small regressions on some public benchmarks, which it reduced by mixing pretraining updates into the RL stage. Supervised fine-tuning teaches the format; the reward model captures what people prefer but could not easily write down; and the RL stage turns those preferences into behavior. The pipeline does not require human labelers: the comparisons can come from another model guided by written principles (Section 22.10).

HISTORICAL NOTE

Learning a reward from human comparisons and optimizing it with reinforcement learning was demonstrated by Christiano et al. (2017) on Atari games and simulated robots, with about an hour of human time per task. Ziegler et al. (2019) applied it to language models, and Stiennon et al. (2020) showed on summarization that policies trained this way were preferred to much larger supervised models and to the human reference summaries. InstructGPT (Ouyang et al. 2022) and Anthropic's helpful-and-harmless assistant (Bai, Jones, et al. 2022) brought the recipe to general-purpose assistants in 2022.

22.3 The KL-regularized optimum

Before choosing an algorithm it pays to ask what the objective's best policy looks like. For (22.1) the answer has a closed form, and it explains much of the rest of the chapter.

The objective is a sum over prompts, and for each prompt it depends only on the distribution $\pi(\cdot\mid x)$. So we can optimize one prompt at a time, over all probability distributions on its responses, and drop x from the notation when it is fixed. Write

$$F(\pi) = \mathbb{E}_{y\sim\pi}[r(y)] - \beta\, D_{\text{KL}}(\pi\,\|\,\pi_{\text{ref}})$$

for the objective of one prompt, with any reward r (a reward model, a checker, or anything else).

Theorem 22.1 The optimal policy of the KL-regularized objective

Let the set of responses $\mathcal{Y}$ be finite, let $r : \mathcal{Y} \to \mathbb{R}$ be any reward and $\beta > 0$. Over all probability distributions π on $\mathcal{Y}$, the objective $F(\pi)$ has the unique maximizer

$$\pi^*(y \mid x) = \frac{1}{Z(x)}\, \pi_{\text{ref}}(y \mid x)\, \exp\big(r(x,y)/\beta\big), \qquad Z(x) = \sum_{y \in \mathcal{Y}} \pi_{\text{ref}}(y \mid x)\, \exp\big(r(x,y)/\beta\big), \tag{22.2}$$

and the optimal value is $F(\pi^*) = \beta \log Z(x)$.

Proof. If π puts positive probability on a response with $\pi_{\text{ref}}(y) = 0$, then $D_{\text{KL}}(\pi \,\|\, \pi_{\text{ref}}) = \infty$ and $F(\pi) = -\infty$, so such π cannot be optimal. For every other π, gather the terms of F under one sum and use $\pi_{\text{ref}}(y) e^{r(y)/\beta} = Z\, \pi^*(y)$:

$$F(\pi) = \sum_y \pi(y) \Big[r(y) - \beta \log \frac{\pi(y)}{\pi_{\text{ref}}(y)} \Big] = -\beta \sum_y \pi(y) \log \frac{\pi(y)}{\pi_{\text{ref}}(y)\, e^{r(y)/\beta}} = -\beta\, D_{\text{KL}}(\pi \,\|\, \pi^*) + \beta \log Z.$$

The KL divergence is nonnegative and is zero only when its two arguments are equal (Gibbs' inequality, Appendix A). So $F(\pi) \le \beta \log Z$, with equality exactly when $\pi = \pi^*$. ∎

In words: start from the reference distribution and multiply the probability of each response by $\exp(r/\beta)$, then renormalize. The optimal policy *reweights* the reference model; it does not replace it. A response that earns one unit more reward than another becomes $e^{1/\beta}$ times more likely relative to it than it was under π_{ref}.

Example 22.2 Four responses

A prompt has four possible responses, A to D, with reference probabilities and rewards

Table 22.1. The prompt of Example 22.2.

Response	A	B	C	D
$\pi_{\text{ref}}(y)$	0.50	0.30	0.15	0.05
$r(y)$	0	1	2	3

The best response, D, is the one the reference model rarely writes. Under π_{ref} the expected reward is $0.3 + 0.3 + 0.15 = 0.75$. With $\beta = 1$ the unnormalized weights $\pi_{\text{ref}}(y) e^{r(y)}$ are 0.500, 0.815, 1.108 and 1.004; they sum to $Z = 3.428$, so

$$\pi^* = (0.146,\ 0.238,\ 0.323,\ 0.293).$$

The expected reward rises to 1.763, at a cost of $D_{\text{KL}}(\pi^* \,\|\, \pi_{\text{ref}}) = 0.531$ nats, and the objective is $1.763 - 0.531 = 1.232 = \log Z$, as the theorem says. Putting all the probability on D would earn 3 but cost $\log(1/0.05) = 3.00$ nats, for an objective of only 0.004.

Changing β moves along a family of policies:

Table 22.2. The KL-optimal policy of Example 22.2 for several values of β; the last column is $D_{\mathrm{KL}}(\pi^* \,\|\, \pi_{\mathrm{ref}})$ in nats.

β	π^*(A)	π^*(B)	π^*(C)	π^*(D)	$\mathbb{E}_{\pi^*}[r]$	KL
∞	0.500	0.300	0.150	0.050	0.750	0
2	0.307	0.304	0.251	0.138	1.219	0.123
1	0.146	0.238	0.323	0.293	1.763	0.531
0.5	0.016	0.071	0.264	0.649	2.546	1.655
0.25	0.000	0.002	0.052	0.946	2.944	2.716

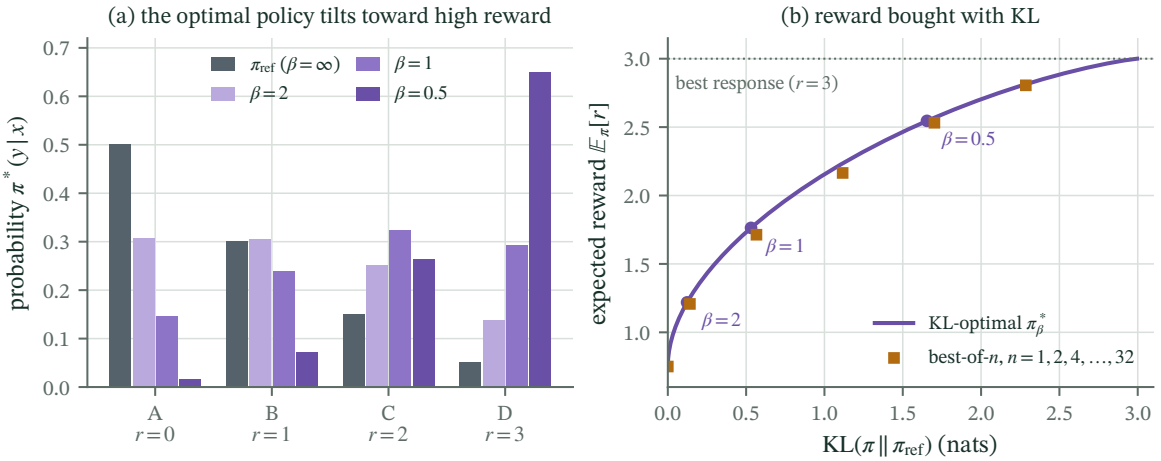


Figure 22.2. The KL-regularized optimum for the prompt of Example 22.2. (a) As β decreases, probability moves from the likely low-reward response A to the rare high-reward response D. (b) Each β gives a point on the reward-KL frontier: no policy earns more reward for the same divergence. The squares are best-of-n sampling (Section 22.9), which lies just below the frontier.

Figure 22.2 shows both views. Several consequences are worth stating.

β is the price of moving. A small β makes the policy approach the reward maximizer; a large β keeps it at the reference. As β varies, the optimal policies trace the *reward–KL frontier* of Figure 22.2b: for every divergence budget, the policy on the frontier earns the most reward that any policy with that divergence can earn (Exercise 22.6). Choosing β is choosing how far to trust the reward.

The support never grows. Since $\pi^*(y) = 0$ whenever $\pi_{\mathrm{ref}}(y) = 0$, KL-regularized optimization can only reweight what the reference model can already produce. This is exact, but it is less restrictive than it sounds, because $\exp(r/\beta)$ can raise a probability of 10^{-12} to something large. Section 22.10 returns to what this means for reasoning.

The policy determines the reward. Taking logarithms in (22.2) and rearranging,

$$r(x, y) = \beta \log \frac{\pi^*(y \mid x)}{\pi_{\text{ref}}(y \mid x)} + \beta \log Z(x).$$

The optimal policy and the reward determine each other, up to a term $\beta \log Z(x)$ that depends on the prompt alone, which is exactly the ambiguity of Bradley–Terry rewards. This observation is the basis of direct preference optimization (Section 22.5).

The reference model is a prior. The formula is Bayes' rule in disguise: π_{ref} is a prior over responses, $\exp(r/\beta)$ plays the part of a likelihood, and π^* is the posterior (Korbak et al. 2022). When π_{ref} is uniform, the KL penalty equals a constant minus the entropy, and (22.1) becomes the maximum-entropy objective of Section 13.5 with temperature β, whose optimum is the softmax $\pi^* \propto e^{r/\beta}$. The formula is also one step of the natural policy gradient in its multiplicative form, $\pi_{k+1} \propto \pi_k \exp(\eta A^{\pi_k})$ (Section 14.4), taken from π_{ref} with step size $\eta = 1/\beta$: the KL penalty acts as a trust region around the reference model (Section 12.4).

The KL divergence decomposes over tokens. The divergence in (22.1) is between distributions over whole responses, which no one can enumerate. Because $\log \pi(y \mid x)$ is a sum over tokens, the divergence is an expectation of a sum of token-level terms:

$$D_{\text{KL}}\big(\pi_\theta(\cdot \mid x) \,\|\, \pi_{\text{ref}}(\cdot \mid x)\big) = \mathbb{E}_{y\sim\pi_\theta}\Big[\sum_t \log \frac{\pi_\theta(y_t \mid s_t)}{\pi_{\text{ref}}(y_t \mid s_t)}\Big] = \mathbb{E}_{y\sim\pi_\theta}\Big[\sum_t D_{\text{KL}}\big(\pi_\theta(\cdot \mid s_t) \,\|\, \pi_{\text{ref}}(\cdot \mid s_t)\big)\Big],$$

with $s_t = (x, y_{<t})$. In words: the divergence between two response distributions is the expected sum, over the positions of a response, of the divergences between their next-token distributions. Exercise 22.7 proves the second equality; it is what makes the penalty computable token by token.

KEY IDEA

The objective "maximize reward minus β times the KL divergence from a reference model" has the closed-form optimum $\pi^* \propto \pi_{\text{ref}}\, e^{r/\beta}$: the reference model, reweighted by exponentiated reward. Algorithms differ in how they approximate this one policy.

CHECKPOINT

1. In Example 22.2, what is $\pi^*(\text{D})/\pi^*(\text{A})$ when $\beta = 1$, and why?
2. What happens to π^* as $\beta \to \infty$, and as $\beta \to 0$?
3. If the reward of every response of a prompt increases by 5, how does π^* change?

Answers.

1. $\frac{0.05\, e^3}{0.5\, e^0} = 0.1\, e^3 \approx 2.01$: the reference ratio 0.1 times $e^{(3-0)/\beta}$.
2. As $\beta \to \infty$ the weights $e^{r/\beta}$ tend to 1 and $\pi^* \to \pi_{\text{ref}}$. As $\beta \to 0$ the response with the highest reward dominates (among those with $\pi_{\text{ref}} > 0$), and π^* tends to the reward maximizer.
3. Not at all: every weight is multiplied by $e^{5/\beta}$, and the normalization cancels it.

22.4 PPO for language models

InstructGPT optimized (22.1) with PPO (Algorithm 12.2) in the token-level MDP, and this remained the standard method for several years. Adapting PPO to language models takes three decisions: where to put the KL penalty, how to estimate advantages, and how to manage the cost.

The KL penalty goes into the reward, token by token. Using the decomposition above, the penalty is spread over the tokens of the response. Token t of a response sampled from the current policy $\pi_{\theta_{\text{old}}}$ receives the reward

$$r_t = -\beta \log \frac{\pi_{\theta_{\text{old}}}(y_t \mid s_t)}{\pi_{\text{ref}}(y_t \mid s_t)} + \mathbb{1}\{t = L\}\, r_\psi(x, y),$$

where L is the length of the response and $\mathbb{1}\{\cdot\}$ is 1 when its condition holds and 0 otherwise. With $\gamma = 1$ the return of the whole response is $r_\psi(x, y) - \beta \log \frac{\pi_{\theta_{\text{old}}}(y|x)}{\pi_{\text{ref}}(y|x)}$, a one-sample estimate of the objective of (22.1) for that prompt (Ziegler et al. 2019; Stiennon et al. 2020). The penalty is computed once, when the response is sampled, and it is then a reward like any other.

A value head and generalized advantage estimation. A critic $V_w(s_t)$ estimates the expected remaining return from the prefix s_t. It is usually a second network of the same size, or a scalar output added to the policy network, and InstructGPT initialized it from the reward model. The TD errors are $\delta_t = r_t + V_w(s_{t+1}) - V_w(s_t)$, with $V_w = 0$ after the last token, and the advantages come from generalized advantage estimation ((11.5)) with $\gamma = 1$: $\hat{A}_t = \sum_{l \geq 0} \lambda^l \delta_{t+l}$. Each token is then one action in PPO's clipped objective ((12.6)).

Algorithm 22.1 PPO for RLHF

Input: reference model π_{ref}, reward model r_ψ, KL strength β, clipping range ϵ, GAE parameter λ, prompts $\mathcal{D}$.

1. Initialize the policy $\pi_\theta \leftarrow \pi_{\text{ref}}$ and the critic V_w (for example from r_ψ).
2. Repeat:
   1. Sample a batch of prompts from $\mathcal{D}$; set $\theta_{\text{old}} \leftarrow \theta$ and generate one or more responses per prompt with $\pi_{\theta_{\text{old}}}$, recording the log-probabilities of $\pi_{\theta_{\text{old}}}$ and π_{ref} for every token.
   2. Score each response with $r_\psi(x, y)$ and form the per-token rewards r_t.
   3. Compute $V_w(s_t)$, the TD errors δ_t, the advantages $\hat{A}_t$ by GAE with $\gamma = 1$, and the targets $\hat{R}_t = \hat{A}_t + V_w(s_t)$.
   4. For a few epochs over minibatches of tokens: ascend PPO's clipped objective in θ, with ratios $\pi_\theta(y_t \mid s_t)/\pi_{\theta_{\text{old}}}(y_t \mid s_t)$; descend $(V_w(s_t) - \hat{R}_t)^2$ in w.
   5. Optionally adjust β to keep the measured KL divergence near a target.

Output: the fine-tuned policy π_θ.

A small example shows the bookkeeping. A three-token response has log-ratios $\log(\pi_{\theta_{\text{old}}}/\pi_{\text{ref}}) =$

0.2, −0.1 and 0.3 at its three tokens, $\beta = 0.1$, and the reward model gives it $r_\psi = 1.5$. The per-token rewards are −0.02, +0.01 and $1.5 - 0.03 = 1.47$, and the returns from each token to the end are 1.46, 1.48 and 1.47; the first equals $1.5 - 0.1 \times 0.4$, the reward minus the penalty for the whole response. If the critic predicts $V_w = 1.2$, 1.4 and 1.5 at the three prefixes, the TD errors are $\delta_1 = -0.02 + 1.4 - 1.2 = 0.18$, $\delta_2 = 0.11$ and $\delta_3 = 1.47 - 1.5 = -0.03$, and with $\lambda = 1$ the advantages are 0.26, 0.08 and −0.03: the critic thinks the first token was a good choice and the last one slightly worse than expected.

The cost. RLHF with PPO keeps four networks of similar size in memory: the policy, the reference model, the reward model and the critic. Generating responses token by token dominates the running time, so large systems run generation and learning on separate machines, and the weights used for generation lag a little behind the learner's. Responses have different lengths, so batches need padding and masks. Learning the critic is itself hard: it must predict, from a prefix, the reward model's score of a response that has not been written yet. Implementation details matter as much here as in Section 12.7; Huang et al. (2024) document dozens of them for one summarization setup. The KL strength can be fixed or adapted: Ziegler et al. (2019) raised β when the measured divergence exceeded a target and lowered it otherwise.

LAB

Lab 22.2: Verifiable rewards in miniature (`labs/ch22_rlvr.py`) trains the toy model of Example 22.1 with token-level PPO, using a tabular critic and the per-token KL penalty above, and compares it with the critic-free methods of Section 22.6. Run `python3 labs/ch22_rlvr.py`; watch the KL divergence from the reference model grow as the reward rises, and how β slows both down.

22.5 Direct preference optimization

The RLHF pipeline needs a reward model, sampling during training, a critic and PPO's many settings. *Direct preference optimization* (DPO) (Rafailov et al. 2023) trains the policy on preference data directly, with a loss as simple as logistic regression. It rests on the observation of Section 22.3 that the optimal policy and the reward determine each other.

The derivation. Start from the optimal policy (22.2) and solve it for the reward:

$$r(x, y) = \beta \log \frac{\pi^*(y \mid x)}{\pi_{\text{ref}}(y \mid x)} + \beta \log Z(x).$$

Substitute this into the Bradley–Terry model. The term $\beta \log Z(x)$ is the same for both responses of a pair, so it cancels:

$$\Pr(y_w \succ y_l \mid x) = \sigma\Big(\beta \log \frac{\pi^*(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi^*(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\Big).$$

The preference probabilities are now written in terms of the optimal policy, with no reward and no partition function in sight. Replace π^* by the policy π_θ being trained and fit it to the preference data by maximum likelihood. The result is the DPO loss:

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}}\Big[\log \sigma\Big(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\Big)\Big]. \tag{22.3}$$

In words: DPO is the Bradley–Terry loss of stage 2, in which the reward of a response is replaced by its *implicit reward* $\hat{r}_\theta(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$. The policy *is* the reward model.

The gradient. Write $h = \hat{r}_\theta(x, y_w) - \hat{r}_\theta(x, y_l)$ for the implicit margin of a pair. Since $\frac{d}{dh}\big[-\log \sigma(h)\big] = -\sigma(-h)$ and $\nabla_\theta h = \beta\big(\nabla_\theta \log \pi_\theta(y_w \mid x) - \nabla_\theta \log \pi_\theta(y_l \mid x)\big)$,

$$\nabla_\theta \mathcal{L}_{\text{DPO}} = -\beta\, \mathbb{E}\big[\sigma(-h)\big(\nabla_\theta \log \pi_\theta(y_w \mid x) - \nabla_\theta \log \pi_\theta(y_l \mid x)\big)\big].$$

A gradient step raises the log-probability of the preferred response and lowers that of the other, weighted by $\sigma(-h)$, the probability that the implicit reward currently ranks the pair the wrong way. Pairs that are already ranked confidently contribute little.

Example 22.3 DPO on four responses

Take the prompt of Example 22.2, $\beta = 1$, and the pair D $\succ$ A. At the start of training $\pi_\theta = \pi_{\text{ref}}$, so both implicit rewards are 0, $h = 0$ and the loss is $-\log \sigma(0) = \log 2 = 0.693$. The weight is $\sigma(0) = 0.5$; for a policy with one logit per response, the gradient of the loss is $+0.5$ on the logit of A and -0.5 on the logit of D, so gradient descent moves probability from A to D.

Now suppose the preferences were generated by the Bradley–Terry model with the rewards of Table 22.1, and evaluate the policy π^* of Example 22.2. Its implicit rewards are $\log(\pi^*/\pi_{\text{ref}}) = (-1.232, -0.232, 0.768, 1.768)$, which are the true rewards shifted by $-\log Z = -1.232$. Every implicit margin equals the true reward difference; for instance $h = 1.768 - (-1.232) = 3 = r(\text{D}) - r(\text{A})$. The model then predicts $\Pr(\text{D} \succ \text{A}) = \sigma(3) = 0.953$, exactly the frequency in the data, and the expected gradient of the DPO loss is zero on every pair: π^* is a minimizer, as the next proposition shows in general.

♦

22.5.1 DPO fits a reward model in disguise ●●

Proposition 22.2 DPO recovers the KL-optimal policy

Fix a prompt with a finite set of responses $\mathcal{Y}$ and a reference model with $\pi_{\text{ref}}(y) > 0$ for all y. Suppose every pair of responses is compared with positive probability, and the label of each comparison follows the Bradley–Terry model with reward r. Then, among all policies with full support, the expected DPO loss is minimized exactly by the KL-optimal policy $\pi^* \propto \pi_{\text{ref}}\, e^{r/\beta}$.

Proof. Write the implicit reward as $\hat{r}(y) = \beta \log(\pi(y)/\pi_{\text{ref}}(y))$. As π ranges over all policies with full support, $\hat{r}$ ranges over all functions on $\mathcal{Y}$, up to an additive constant: given any function g, the policy $\pi \propto \pi_{\text{ref}}\, e^{g/\beta}$ has $\hat{r} = g - \beta \log \sum_y \pi_{\text{ref}}(y) e^{g(y)/\beta}$. The expected DPO loss is the expected Bradley–Terry log-loss of the scores $\hat{r}$. For each compared pair (y, y') it is the cross-entropy between the true preference probability $\sigma(r(y) - r(y'))$ and the predicted one $\sigma(\hat{r}(y) - \hat{r}(y'))$, which is minimized exactly when the two are equal, that is, when $\hat{r}(y) - \hat{r}(y') = r(y) - r(y')$. Every pair is compared, so the minimum over all pairs is attained exactly when $\hat{r} - r$ is constant on $\mathcal{Y}$. Then $\pi \propto \pi_{\text{ref}}\, e^{\hat{r}/\beta} \propto \pi_{\text{ref}}\, e^{r/\beta}$, which is π^*. ■

The proposition describes an idealized situation: infinite data, every pair compared, preferences that really follow a Bradley–Terry model, and a policy class rich enough to represent π^*. Each assumption fails in practice, and each failure explains a known weakness.

DPO learns offline. It uses only the responses in the data set, usually written by π_{SFT} or by other models, and it never tries its own. Nothing in the loss controls the probability of responses that appear in no pair, so the policy can move mass to unseen responses in ways the data never sanction. A careful comparison found that well-tuned PPO outperformed DPO on the tasks it studied, and argued that DPO can exploit responses outside the distribution of the preference data (Xu et al. 2024). Iterative or *online* versions regenerate responses with the current policy, collect new preferences and repeat, which moves DPO back toward reinforcement learning.

Deterministic preferences break the regularization. If y_w beats y_l in every comparison, the Bradley–Terry likelihood is maximized only as the margin h grows without bound, whatever β is. DPO then pushes $\pi_\theta(y_l)$ toward zero, although the KL-optimal policy of any finite reward gives y_l positive probability (Azar et al. 2024). In the four-response example with the single pair $\mathsf{D} \succ \mathsf{A}$, gradient descent on the DPO loss lowers $\pi_\theta(\mathsf{A})$ from 0.5 to 0.002 after 2,000 steps and to 0.0002 after 20,000, with no limit in sight. The *IPO* loss of Azar et al. (2024) replaces $-\log \sigma(h)$ by a squared loss $\left(h' - \frac{1}{2\tau}\right)^2$, where $h' = h/\beta$ is the margin of log-ratios and $\tau > 0$ is a regularization strength; its minimizers have the finite margin $h' = 1/(2\tau)$ (Exercise 22.9).

The loss sees only differences. The DPO loss depends on the margin, not on the two log-probabilities separately, so both can fall during training. Several groups have observed that the probability of the preferred responses decreases while DPO improves the margin, and Razin et al. (2025) analyze this *likelihood displacement*, which can move probability to responses that are worse than either member of the pair.

22.5.2 Variants

DPO started a family of direct alignment methods, of which we mention a few. *IPO* (Azar et al. 2024) is the squared-loss version just described. *KTO* (Ethayarajh et al. 2024) learns from unpaired labels ("this response is good", "this one is bad") instead of pairs, using a loss inspired by prospect theory with a reference point estimated from the KL divergence. *SimPO* (Meng et al. 2024) drops the reference model and uses the average log-probability per token as the implicit reward, with a target margin; *ORPO* (Hong et al. 2024) adds an odds-ratio penalty to the supervised fine-tuning loss, also without a reference model. When preferences are not transitive, no reward of any kind can represent them, and *Nash learning from human feedback* (Munos et al. 2024) instead seeks a policy that is preferred to every other policy at least half of the time.

LAB

Lab 22.1: Preferences and RLHF in miniature (`labs/ch22_llm.py`). Run `python3 labs/ch22_llm.py`. You will compute the KL-optimal policy of Example 22.2 and check that REINFORCE with a KL penalty converges to it, fit a Bradley–Terry reward model to synthetic preferences, train a policy with DPO on the same data and compare it with the KL-optimal policy of the fitted reward, and watch DPO drive a deterministically rejected response to zero probability while IPO does not.

22.6 Reinforcement learning with verifiable rewards

Some tasks come with a checker. The final answer of a mathematics problem can be compared with the known answer; a program can be compiled and run against unit tests; a formatting rule ("put the final answer in a box") can be verified by a script. When such a check exists, it can serve as the reward directly, with no reward model. This is *reinforcement learning with verifiable rewards* (RLVR), a name introduced with the Tülu 3 models (Lambert et al. 2024). Earlier work trained learned verifiers for mathematics problems (Cobbe, Kosaraju, et al. 2021), and code-generation models were evaluated by running their programs (M. Chen et al. 2021).

Verifiable rewards change the character of the problem. The reward is usually binary and arrives at the end of a response that may be thousands of tokens long, because the model is encouraged to reason step by step before answering, writing a *chain of thought* (Wei et al. 2022). Prompts differ widely in difficulty: some are solved by almost every sample, others by none. And a critic that predicts, from a prefix of a long derivation, the probability that the derivation will end with a correct answer is both hard to learn and as large as the policy. These facts favor methods that use no critic at all.

22.6.1 GRPO

Group relative policy optimization (GRPO) (Shao et al. 2024) replaces the critic with a group of samples. For each prompt it samples a *group* of G responses $y_1, \dots, y_G$ from the current policy, scores them, and measures each response against the others: the advantage of response i is its reward standardized within the group,

$$\hat{A}_i = \frac{r_i - \operatorname{mean}(r_1, \dots, r_G)}{\operatorname{std}(r_1, \dots, r_G)},$$

and every token of y_i receives this same advantage. (Implementations add a small constant to the denominator; they differ in whether the standard deviation divides by G or by $G-1$, which rescales the advantages of a group by a constant.) The group mean acts as a Monte Carlo estimate of the value of the prompt, the baseline of Section 11.5, computed from samples instead of predicted by a network. The policy then ascends a PPO-style clipped objective with a KL penalty:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}\Big[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\Big(\min\big(\rho_{i,t}\hat{A}_i,\ \operatorname{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\,\hat{A}_i\big) - \beta\,\hat{k}_{i,t}\Big)\Big],$$

where $|y_i|$ is the number of tokens of response i, $\rho_{i,t} = \pi_\theta(y_{i,t} \mid s_{i,t}) / \pi_{\theta_{\text{old}}}(y_{i,t} \mid s_{i,t})$ is the probability ratio of its token t in the state $s_{i,t} = (x, y_{i,<t})$, and $\hat{k}_{i,t}$ is a per-token estimate of the KL divergence from π_{ref} (the estimator k_3 of Section 22.7). Unlike PPO for RLHF, GRPO puts the KL penalty in the loss rather than in the reward.

Algorithm 22.2 GRPO

Input: reference model π_{ref}, a reward function $r(x, y)$ (for example a checker), group size G, clipping range ϵ, KL strength $\beta \geq 0$, step size η, number of inner updates μ, prompts $\mathcal{D}$.

1. Initialize $\pi_\theta \leftarrow \pi_{\text{ref}}$.
2. Repeat:
   1. Sample a batch of prompts from $\mathcal{D}$ and set $\theta_{\text{old}} \leftarrow \theta$.
   2. For each prompt x in the batch: sample G responses $y_1, \dots, y_G \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$ and compute their rewards $r_i = r(x, y_i)$.
   3. For each group: compute $\hat{A}_i = (r_i - \text{mean}(r))/(\text{std}(r) + \varepsilon_0)$ and assign it to every token of y_i; here ε_0 is a small constant.
   4. Repeat μ times: $\theta \leftarrow \theta + \eta \nabla_\theta J_{\text{GRPO}}(\theta)$, estimated on the sampled groups (in minibatches).

Output: the fine-tuned policy π_θ.

Step 3 has a consequence that matters a great deal in practice: if all G responses of a prompt receive the same reward, every advantage of the group is zero, and the group contributes nothing to the policy update except through the KL term. A prompt the model always solves teaches nothing, and so does a prompt it never solves.

Example 22.4 Group-relative advantages

A group of $G = 8$ responses to a prompt has rewards $(1, 0, 0, 1, 0, 0, 0, 0)$. The mean is 0.25 and the standard deviation is $\sqrt{0.25 \times 0.75} = 0.433$, so the two correct responses get $\hat{A} = 0.75/0.433 = 1.732$ and the six incorrect ones get $\hat{A} = -0.25/0.433 = -0.577$. The advantages sum to zero. In general, for binary rewards with a fraction p of correct responses in the group, correct responses get $\sqrt{(1-p)/p}$ and incorrect ones $-\sqrt{p/(1-p)}$: the rarer success is, the more a success is worth.

Clipping works as in PPO. Suppose that after one inner update the ratio of a token of a correct response has risen to $\rho = 1.3$, with $\epsilon = 0.2$. The clipped term 1.2×1.732 is smaller than 1.3×1.732, so the minimum selects it, and it does not depend on θ: that token's gradient is zero until the next batch is sampled.

Figure 22.3 shows four groups. With one success out of eight, that success receives $\hat{A} = 2.65$; with seven out of eight, the single failure receives -2.65; a group of eight failures receives nothing.

♦

22.6.2 REINFORCE with a leave-one-out baseline

GRPO's group is an old idea in new clothes. With k samples per prompt, the *leave-one-out* baseline (Kool et al. 2019) scores each sample against the average of the *other* $k - 1$:

$$\hat{A}_i^{\text{RLOO}} = r_i - \frac{1}{k-1} \sum_{j \neq i} r_j.$$

The baseline of sample i does not depend on y_i, so by Theorem 11.2 the estimate

$$\frac{1}{k} \sum_{i=1}^{k} \hat{A}_i^{\text{RLOO}} \nabla_\theta \log \pi_\theta(y_i \mid x)$$

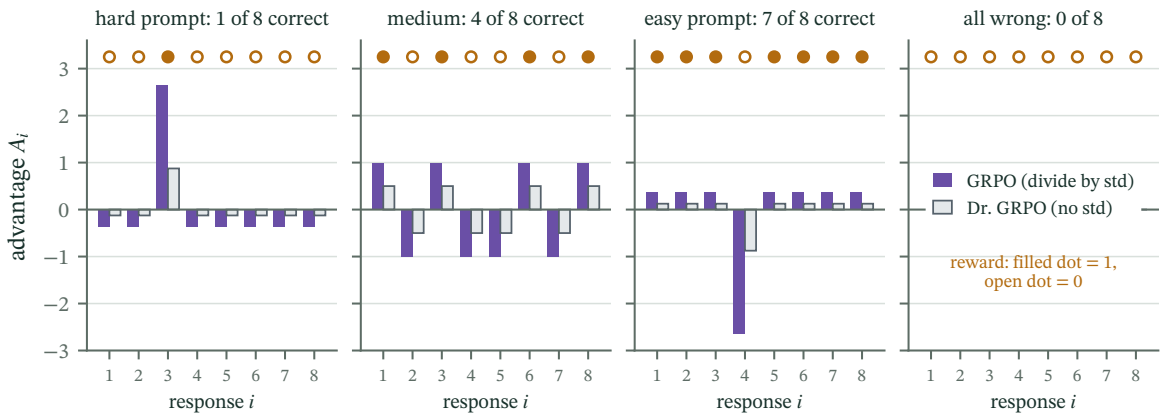


Figure 22.3. Group-relative advantages for four groups of eight binary rewards (dots: filled for reward 1, open for 0). GRPO divides by the group's standard deviation; Dr. GRPO, discussed below, does not. Rare outcomes get large advantages, and a group whose rewards are all equal gets none.

is an unbiased estimate of the policy gradient (Exercise 22.12). This *RLOO* estimator is REINFORCE (Algorithm 11.1) at the level of whole responses, and Ahmadian et al. (2024) argued that it suits RLHF better than PPO: the starting policy is already good, the reward arrives once per response, and the machinery of PPO for small, safe steps buys little. In their experiments RLOO outperformed PPO at lower cost.

The relation to GRPO is exact up to the standard deviation. A short computation shows $r_i - \text{mean}(r) = \frac{k-1}{k}\hat{A}_i^{\text{RLOO}}$, so centering by the group mean *including* sample i gives an unbiased gradient shrunk by the constant factor $\frac{k-1}{k}$. Dividing by the group's standard deviation, which depends on r_i, breaks this exactness, as we will see.

22.6.3 GRPO on a toy task

Figure 22.4 runs GRPO on the digit-sum task of Example 22.1, with six prompts asking for the targets 3, 5, 7, 9, 11 and 13. The reference model has random logits and starts with success probabilities between 5.8% (target 3) and 0.44% (target 13). Each iteration samples a group of $G = 8$ responses per prompt; there is no KL penalty.

Two things stand out. First, the prompts are learned roughly in order of how often the reference model solved them. A prompt starts to improve only after the model happens to produce a correct response for it, so the hard prompt (target 13, initial success 0.44%) finishes at 2.1% on average after 300 iterations, while the five easier prompts end between 93% and 99.5%. Reinforcement learning with verifiable rewards depends on the model already being able to succeed sometimes. Second, on average 71% of the groups carry no learning signal at all: at first because every response failed, and later because every response succeeded. Real systems face the same waste, and one of the fixes below addresses it.

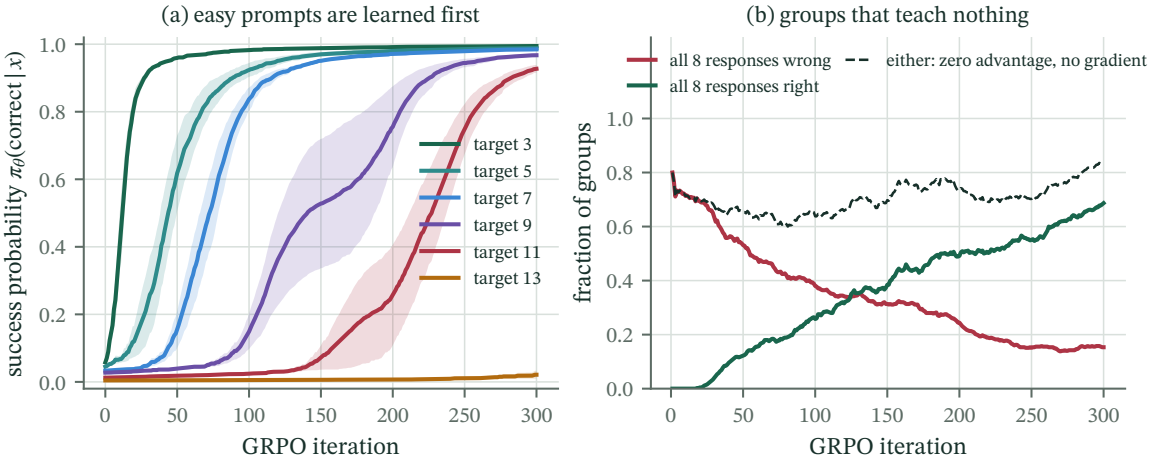


Figure 22.4. GRPO on the digit-sum task (mean over 5 seeds; bands show ±1 standard error). (a) Exact success probabilities: prompts are learned in order of their initial success rate, and target 13 is not learned within 300 iterations. (b) Fraction of groups in which all eight responses had the same reward (moving average over 15 iterations): early on, most groups fail completely; later, most succeed completely. Either way they produce no gradient.

22.6.4 Reasoning models and DeepSeek-R1

In 2024 OpenAI reported that its o1 models, trained with large-scale reinforcement learning to reason before answering, improved steadily both with more reinforcement-learning compute during training and with more "thinking" time at test time (OpenAI 2024). Few details were published. The DeepSeek-R1 work (Guo et al. 2025) described an open recipe, and its findings, as reported, shaped much of the field in 2025.

DeepSeek-R1-Zero applied GRPO directly to a pretrained base model, without any supervised fine-tuning, with rule-based rewards: an accuracy reward from checking final answers (or running code) and a format reward for placing the reasoning between designated tags. The authors reported that accuracy on the 2024 American Invitational Mathematics Examination (AIME) rose during training from about 16% to over 70% (pass@1, the probability that a single sample is correct), that the average response length grew steadily to thousands of tokens without any reward for length, and that behaviors such as re-checking earlier steps and trying alternative approaches became frequent. They also reported weaknesses: poor readability and mixing of languages within a response.

DeepSeek-R1 added stages around the same core: supervised fine-tuning on a small set of long, readable chains of thought; reasoning-oriented RL with an extra reward for consistent language; rejection sampling from the resulting model to build a large supervised data set, mixed with data for non-reasoning tasks; and a final RL stage that combined rule-based rewards with learned reward models for helpfulness and harmlessness. The authors also fine-tuned smaller open models on outputs of DeepSeek-R1 and reported that this distillation worked better than applying RL to the small models directly.

These are reports from one group about one family of models, and open replications quickly

probed them. Z. Liu et al. (2025) found that the base models already show self-reflection before any RL, and argued that part of the length growth comes from a bias in GRPO's normalization rather than from better reasoning; the next subsection explains it. What many independent reports agree on is that RL with verifiable rewards improves pass@1 on mathematics and code benchmarks for strong base models. How much of the improvement is new capability rather than better use of existing capability is debated (Section 22.10).

22.6.5 The biases of GRPO's normalizations

GRPO normalizes twice, once over the tokens of a response ($1/|y_i|$) and once over the rewards of a group (dividing by the standard deviation). Both normalizations change which direction the update takes, not only its size.

The length bias. Summing GRPO's token terms for one response (with $\rho = 1$, at the first inner step) gives $\frac{\hat{A}_i}{G|y_i|}\sum_t \nabla_\theta \log \pi_\theta(y_{i,t} \mid s_{i,t}) = \frac{\hat{A}_i}{G|y_i|}\nabla_\theta \log \pi_\theta(y_i \mid x)$. So GRPO weights each response's log-probability gradient by $\hat{A}_i/|y_i|$, whereas the policy gradient with a group baseline weights it by $\hat{A}_i$. Among correct responses, short ones are pushed up more; among incorrect responses, long ones are pushed down *less*. Consider a group of four with rewards $(1, 1, 0, 0)$, lengths 50, 150, 50 and 250 tokens, and advantages $(1, 1, -1, -1)$. GRPO weights the four log-probability gradients by 0.005, 0.0017, −0.005 and −0.001: the 250-token wrong answer is pushed down five times less than the 50-token wrong answer. Z. Liu et al. (2025) argue that this bias makes incorrect responses grow longer during training. Their *Dr. GRPO* divides every response by the same constant (such as the maximum length) instead of by its own length. *DAPO* (Yu et al. 2025) averages over all the tokens of the group (or batch), which treats the responses of a group equally.

The difficulty bias. For binary rewards with success fraction p, the total weight $\sum_i |\hat{A}_i|$ of a group is $2G\sqrt{p(1-p)}$ with standard-deviation scaling and $2Gp(1-p)$ without it. Their ratio $1/\sqrt{p(1-p)}$ is smallest at $p = \frac{1}{2}$ and grows toward $p = 0$ and $p = 1$, so dividing by the standard deviation gives more weight to prompts that are almost always solved or almost never solved, relative to the medium prompts where the learning signal is richest (Figure 22.3). Dr. GRPO drops the division and uses $r_i - \text{mean}(r)$, which is RLOO up to the factor $\frac{G-1}{G}$.

Wasted groups, exploration and long responses. DAPO (Yu et al. 2025) combines four changes. *Dynamic sampling* keeps sampling prompts and discards those whose groups are all right or all wrong until the batch is full of informative groups, the problem of Figure 22.4b. *Clip-higher* uses a larger upper clipping range than lower one ($\epsilon_{\text{high}} = 0.28$, $\epsilon_{\text{low}} = 0.2$), because a symmetric clip limits how fast unlikely tokens can gain probability and was observed to make the policy's entropy collapse. *Token-level averaging* is the length fix above. *Overlong shaping* applies a soft penalty to responses that approach the length limit, instead of treating every truncated response as wrong. DAPO also dropped the KL penalty, on the grounds that long reasoning should be allowed to move far from the initial model, and reported 50 points on AIME 2024 with a 32-billion-parameter base model (Qwen2.5-32B), more than the 47 reported for DeepSeek-R1-Zero's recipe on the same model, in half as many training steps.

PITFALL

A verifier is not ground truth. Final-answer checkers accept a correct answer reached by wrong reasoning, and can be fooled by unexpected formats; unit tests check some behaviors and not others. Capable models trained against tests have been observed to special-case or edit the tests rather than fix the code (Baker et al. 2025). Verifiable rewards reduce reward hacking but do not eliminate it.

LAB

Lab 22.2: Verifiable rewards in miniature (`labs/ch22_rlvr.py`). Run `python3 labs/ch22_rlvr.py` to train the digit-sum model with GRPO, RLOO and PPO, compare GRPO's normalizations with those of Dr. GRPO and DAPO, add dynamic sampling, and measure pass@k before and after training. Look for the zero-advantage groups of Figure 22.4 and for the prompt that is never learned.

22.7 KL estimators ●●

The KL divergence in (22.1) is a sum over all possible responses, which can never be computed. Every algorithm in this chapter therefore estimates it from the responses it samples, and the choice of estimator matters more than one might expect. This section compares the three estimators in common use (Schulman 2020), first as numbers and then as terms in a loss.

Let $y \sim \pi_\theta(\cdot \mid x)$ be a response sampled from the current policy, and write the log-ratio and the ratio as

$$u = \log \frac{\pi_\theta(y \mid x)}{\pi_{\text{ref}}(y \mid x)}, \qquad \rho = \frac{\pi_{\text{ref}}(y \mid x)}{\pi_\theta(y \mid x)} = e^{-u}.$$

The three per-sample estimators of $D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}) = \mathbb{E}[u]$ are

$$k_1 = u = -\log \rho, \qquad k_2 = \tfrac{1}{2}(\log \rho)^2, \qquad k_3 = (\rho - 1) - \log \rho.$$

Proposition 22.3 Properties of the KL estimators

Let π_θ and π_{ref} be distributions on a finite set with $\pi_\theta(y) > 0$ wherever $\pi_{\text{ref}}(y) > 0$, and let $y \sim \pi_\theta$.

1. k_1 and k_3 are unbiased: $\mathbb{E}[k_1] = \mathbb{E}[k_3] = D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})$.
2. $k_2 \geq 0$ and $k_3 \geq 0$ for every sample, with equality exactly when $\rho = 1$; k_1 is negative whenever $\pi_\theta(y) < \pi_{\text{ref}}(y)$.
3. k_2 is biased: $\mathbb{E}[k_2] - D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}) = \frac{1}{6}\mathbb{E}[u^3] + O(\mathbb{E}[u^4])$ when the log-ratios are small.
4. The variance of k_3 involves $\mathbb{E}[\rho^2] = \sum_y \pi_{\text{ref}}(y)^2/\pi_\theta(y)$, which is unbounded as $\pi_\theta(y) \to 0$ for a response with $\pi_{\text{ref}}(y) > 0$.

Proof. (1) $\mathbb{E}[k_1] = \mathbb{E}[u]$ is the definition of the KL divergence. For k_3, $\mathbb{E}[\rho] = \sum_y \pi_\theta(y) \frac{\pi_{\text{ref}}(y)}{\pi_\theta(y)} = \sum_y \pi_{\text{ref}}(y) = 1$, where the sum runs over the support of π_θ, which contains that of π_{ref}; hence $\mathbb{E}[k_3] = \mathbb{E}[\rho - 1] + \mathbb{E}[u] = D_{\text{KL}}$. (2) k_2 is a square. For k_3, the inequality $\log z \leq z - 1$ for $z > 0$, with equality only at $z = 1$, gives $(\rho - 1) - \log \rho \geq 0$.

(3) The identity $\mathbb{E}[e^{-u}] = \mathbb{E}[\rho] = 1$ and the expansion $e^{-u} = 1 - u + \frac{u^2}{2} - \frac{u^3}{6} + O(u^4)$ give $\mathbb{E}[u] = \frac{1}{2}\mathbb{E}[u^2] - \frac{1}{6}\mathbb{E}[u^3] + O(\mathbb{E}[u^4])$, that is, $\mathbb{E}[k_2] = \frac{1}{2}\mathbb{E}[u^2] = D_{\mathrm{KL}} + \frac{1}{6}\mathbb{E}[u^3] + O(\mathbb{E}[u^4])$.
(4) $\mathrm{Var}(k_3)$ contains the term $\mathbb{E}[\rho^2]$, computed directly from the definition.

■

Near the reference model, where u is small, $k_3 = \frac{u^2}{2} - \frac{u^3}{6} + \cdots$ and $k_2 = \frac{u^2}{2}$ almost agree, and both are of the size of the divergence itself. By contrast $k_1 = u$ fluctuates on the scale $\sqrt{\mathbb{E}[u^2]} \approx \sqrt{2\,D_{\mathrm{KL}}}$, much larger than the KL divergence when the KL is small. So k_1 is unbiased but noisy, k_2 is quiet but slightly biased, and k_3 is unbiased and quiet near the reference, and noisy far from it.

For the policy π^* of Example 22.2 with $\beta = 1$, the true divergence is 0.531. The four responses have $k_1 = (-1.232, -0.232, 0.768, 1.768)$, two of them negative, and $k_3 = (1.196, 0.029, 0.232, 0.939)$. Under sampling from π^*, k_1 has mean 0.531 and standard deviation 1.029, k_2 has mean 0.670 (biased upward by 0.139) and standard deviation 0.615, and k_3 has mean 0.531 and standard deviation 0.449. Figure 22.5 repeats the comparison for a whole family of policies that move away from their reference model.

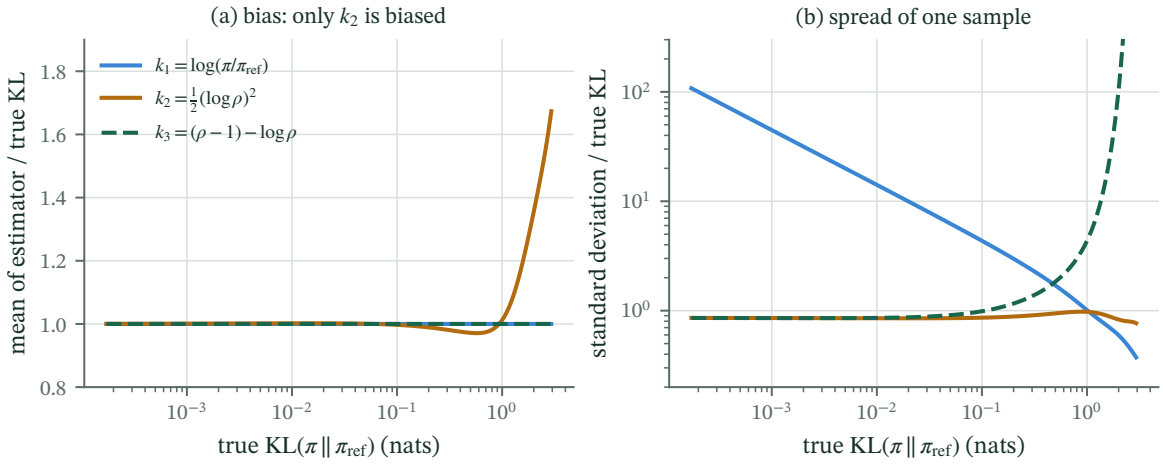


Figure 22.5. The three KL estimators for a prompt with 50 responses, along the family of KL-optimal policies π^*_β (exact values by enumeration). (a) k_1 and k_3 are unbiased; the bias of k_2 appears once the divergence exceeds a fraction of a nat. (b) One sample of k_1 is very noisy relative to a small divergence; k_3 is the quietest near the reference but its spread explodes when the policy is far from it, because $\rho = \pi_{\mathrm{ref}}/\pi_\theta$ can be huge.

Per-token estimators. In practice the estimators are applied to each token, with $\rho_t = \pi_{\mathrm{ref}}(y_t \mid s_t)/\pi_\theta(y_t \mid s_t)$, and summed over the response. For k_1 the sum is exactly the response-level log-ratio. For k_3 the sum is a different estimator from the response-level k_3, but it is still unbiased: conditionally on the prefix s_t, the term of token t is an unbiased estimate of the divergence between the next-token distributions, and by the decomposition of Section 22.3 these divergences add up to the response-level divergence.

In a loss or in the reward. An estimator can enter training in two ways. PPO for RLHF puts k_1 in

the *reward* (Section 22.4): the penalty multiplies the score function, as any reward does. GRPO puts k_3 in the *loss*: it adds βk_3 to the loss (that is, subtracts it from the objective J_{GRPO}) and differentiates it with respect to θ, holding the sampled response fixed. These are not equivalent, and the second does not always give the gradient one might expect.

Proposition 22.4 Which gradient each estimator gives

Let π_θ have full support, let $y \sim \pi_\theta$, and differentiate each estimator with respect to θ with the sample y held fixed. Then

$$\mathbb{E}\big[\nabla_\theta k_1\big] = 0, \qquad \mathbb{E}\big[\nabla_\theta k_2\big] = \nabla_\theta D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}), \qquad \mathbb{E}\big[\nabla_\theta k_3\big] = \nabla_\theta D_{\text{KL}}(\pi_{\text{ref}} \,\|\, \pi_\theta),$$

while k_1 placed in the reward gives $\mathbb{E}\big[k_1 \nabla_\theta \log \pi_\theta(y)\big] = \nabla_\theta D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})$.

Proof. Write $\psi = \nabla_\theta \log \pi_\theta(y)$ for the score; its mean is zero, $\mathbb{E}[\psi] = \sum_y \nabla_\theta \pi_\theta(y) = \nabla_\theta 1 = 0$. The gradient of the divergence is

$$\nabla_\theta \sum_y \pi_\theta(y)\, u(y) = \sum_y \pi_\theta(y)\, \psi(y)\, u(y) + \sum_y \pi_\theta(y)\, \psi(y) = \mathbb{E}[u\psi],$$

since $\nabla_\theta u = \psi$. Now $\nabla_\theta k_1 = \psi$, with mean zero. $\nabla_\theta k_2 = u\,\psi$, with mean $\mathbb{E}[u\psi] = \nabla_\theta D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})$. And $\nabla_\theta k_3 = \nabla_\theta(e^{-u} - 1 + u) = (1 - \rho)\,\psi$, with mean $\mathbb{E}[\psi] - \sum_y \pi_{\text{ref}}(y)\, \nabla_\theta \log \pi_\theta(y) = \nabla_\theta\big(-\sum_y \pi_{\text{ref}}(y) \log \pi_\theta(y)\big) = \nabla_\theta D_{\text{KL}}(\pi_{\text{ref}} \,\|\, \pi_\theta)$, because $D_{\text{KL}}(\pi_{\text{ref}} \,\|\, \pi_\theta)$ equals $-\sum_y \pi_{\text{ref}}(y) \log \pi_\theta(y)$ plus a term that does not depend on θ. Finally $\mathbb{E}[k_1\psi] = \mathbb{E}[u\psi]$. ■

The consequences are practical. Adding k_1 to a loss does nothing in expectation and only adds noise: k_1 belongs in the reward. Adding k_3 to a loss, as GRPO does, pushes the policy along the gradient of the *forward* divergence $D_{\text{KL}}(\pi_{\text{ref}} \,\|\, \pi_\theta)$, not the reverse divergence $D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})$ of the objective. The two divergences agree to second order near π_{ref}, so k_3 in the loss is a sensible regularizer; it is just not the gradient of the objective that is usually written down (Tang and Munos 2025). Two further caveats apply to all three estimators when they are used per token in a loss. The gradient then flows only through the probability of each token, not through that token's effect on the distribution of later prefixes, so even k_2 misses terms: the exact gradient of the response-level divergence weights the score of token t by the sum of the log-ratios of tokens $t, t+1, \ldots$ (the "KL to go"), not by the log-ratio of token t alone. And when several updates reuse the same samples, the samples come from $\pi_{\theta_{\text{old}}}$ rather than π_θ, and the expectations above need importance weights. The lab checks the proposition by enumeration.

22.8 Reward hacking and overoptimization

Every reward in this chapter is a proxy: a reward model approximates human judgment, and a checker approximates correctness. *Goodhart's law*, often summarized as "when a measure becomes a target, it ceases to be a good measure", describes what happens when a proxy is optimized hard, and it comes in several variants: a proxy that is merely noisy rewards lucky errors, and a proxy that is accurate on typical cases can be badly wrong on extreme ones (Manheim and

Garrabrant 2018). The behaviors that exploit a proxy are called *reward hacking* (Amodei et al. 2016).

Language models supply many examples. Reward models tend to prefer longer responses, and RLHF makes responses longer; Singhal et al. (2023) found that much of the measured improvement of RLHF in their settings could be traced to length. Models trained on human preferences tend toward *sycophancy*, agreeing with the views a user expresses even when they are wrong, partly because people rate agreement favorably (Sharma et al. 2024). With verifiable rewards, models exploit loose answer checkers and, as noted above, special-case or edit unit tests (Baker et al. 2025).

Skalse et al. (2022) make the idea precise. A proxy reward is *hackable*, relative to a true reward and a set of policies, if some change of policy increases the proxy's expected return while decreasing the true one. They show that over the set of all stochastic policies, two reward functions can be unhackable only if one of them is constant. Unhackable proxies are thus essentially nonexistent, and the practical question becomes how far a proxy can be optimized before it starts to mislead.

Scaling laws for overoptimization. Gao et al. (2023) studied this question in a synthetic setup where a large "gold" reward model plays the part of human judgment and labels the data used to train smaller proxy reward models. Optimizing a proxy with best-of-n sampling or with PPO, they measured the gold reward as a function of $d = \sqrt{D_{\mathrm{KL}}(\pi \,\|\, \pi_{\mathrm{ref}})}$. The gold reward first rose and then fell, and simple formulas fitted the curves: $d(a - b\,d)$ for best-of-n and $d(a - b \log d)$ for reinforcement learning, where a and b are fitted constants that depend on the method and change smoothly with the size of the proxy reward model; larger proxies overoptimized less. Stiennon et al. (2020) had already observed the same rise and fall with human evaluations of summaries, and Bai, Jones, et al. (2022) reported that the RL reward grew roughly linearly in $\sqrt{D_{\mathrm{KL}}}$.

Example 22.5 A reward model that learned to like length

The overoptimization toy of the lab has 3,000 possible responses, each with a quality $q \sim \mathcal{N}(0, 1)$ and a length between 1 and 40 tokens. The reference model writes short responses (mean length 6.2; only 7% of its probability is on responses longer than 10 tokens). The *gold* reward, standing for what people actually want, rewards quality, and rewards length up to 10 tokens and penalizes it beyond: $r_{\mathrm{gold}} = q + 0.25 \min(\ell, 10) - 0.25 \max(\ell - 10, 0)$, where ℓ is the length. A linear Bradley–Terry reward model sees a noisy measurement of quality and the exact length. Fitted to 5,000 preference pairs sampled from π_{ref} and labeled by the gold reward, it assigns weight 0.73 to its quality score and 0.18 to each token of length: in the data, longer responses really were better.

Figure 22.6 shows the KL-optimal policies for this proxy. As β decreases, the proxy reward keeps rising. The gold reward rises from 1.48 to a peak of 3.73 at $D_{\mathrm{KL}} = 4.4$ nats, where the mean length is 12 tokens, and then falls, to 2.32 at 16 nats, where the policy writes 20 tokens on average. Best-of-n sampling with the proxy behaves the same way, peaking near $n = 512$. The policies that are optimal for the gold reward itself reach 4.6 at the same divergence, without growing past the best length.

♦

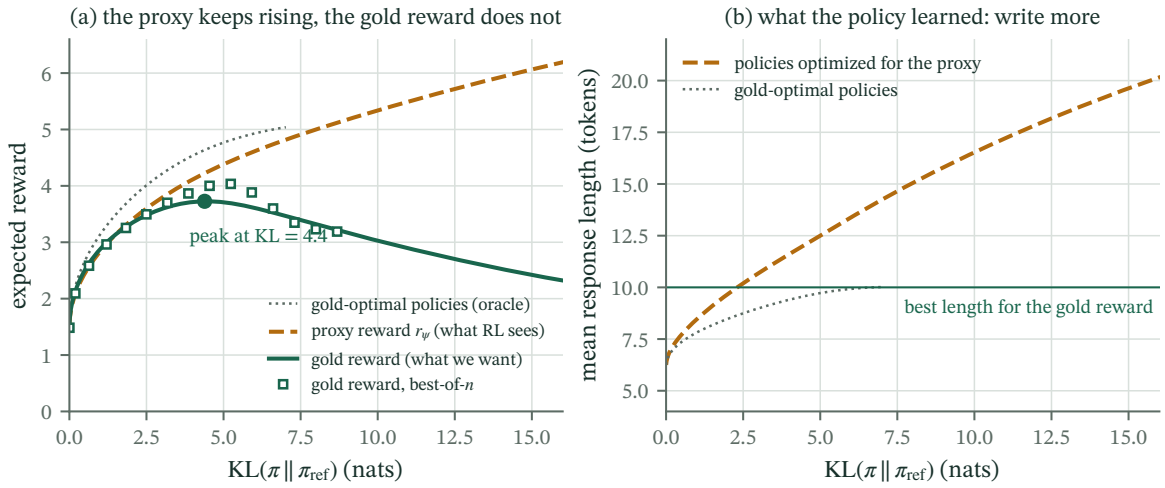


Figure 22.6. Overoptimization of a learned reward in the toy of Example 22.5 (exact computation). (a) Along the KL-optimal policies for the proxy reward model, the proxy (shifted to agree with the gold reward under π_{ref}; Bradley–Terry rewards are defined only up to a constant) rises without limit, while the gold reward peaks and falls. Best-of-n with the proxy shows the same shape. (b) The mechanism: the reward model learned that longer is better, and the optimized policy writes far longer responses than the gold reward wants.

The toy shows two lessons that hold far beyond it. First, the reward model was right where it was trained: on the short responses of π_{ref}, more length really was better. It was wrong only where the data were thin, and optimization went straight there. Second, more preference data would not fix it, because the reward model's linear form cannot represent "up to a point"; overoptimization comes from errors that are systematic, not only from noise.

Mitigations. None of these removes the problem, and they are usually combined.

- *A KL budget and early stopping.* The KL penalty and the number of training steps limit how far the policy moves; choose them by measuring a signal the policy was not trained on, such as human evaluation or a held-out reward model.
- *Better reward models.* Larger reward models trained on more data overoptimize later (Gao et al. 2023). Ensembles of reward models, combined conservatively (for example by their minimum), reduce overoptimization (Coste et al. 2024), but do not eliminate it when all members share the same error (Eisenstein et al. 2023).
- *Fresh data.* Collect new comparisons of the current policy's responses and retrain the reward model, so that it is accurate where the policy now is (Bai, Jones, et al. 2022).
- *Constraints on known exploits.* Penalize length directly, or evaluate with length-controlled comparisons; treat the KL budget, length or other side effects as constraints (compare Section 23.7).
- *Better rewards.* Verifiable rewards (Section 22.6) and rewards for intermediate steps (Section 22.9) where they exist; reward design in general is the subject of Section 23.6.

22.9 Process versus outcome rewards ●●

A chain of thought of 5,000 tokens with a wrong step at token 300 earns the same reward 0 as one that is wrong from the start. An *outcome reward* (a final-answer check, or an *outcome reward model* trained to predict it (Cobbe, Kosaraju, et al. 2021)) says nothing about which steps went wrong. A *process reward model* (PRM) scores every step of a solution, and so offers denser credit assignment.

Lightman et al. (2024) collected PRM800K, a data set of about 800,000 human labels on individual steps of solutions to MATH problems, and compared process and outcome supervision for training verifiers that select the best of many sampled solutions. The process-supervised verifier was the better selector; with it, best-of-n selection solved 78% of the problems of a representative subset of the MATH test set. An earlier study on grade-school word problems (Uesato et al. 2022) found that outcome and process supervision gave similar final-answer accuracy, but that process-based feedback was needed to make the intermediate reasoning correct as well.

Process labels are values. Human step labels are expensive. A cheap alternative labels a step by rolling out many completions from the prefix that ends with it and recording the fraction that reach the correct answer (Wang et al. 2024). That fraction is a Monte Carlo estimate of the value $V^{\pi}(s)$ of the prefix s under the completion policy π, so a PRM trained on such labels is a value function, and the difference between the values after and before a step is an estimate of the step's advantage. This connects process rewards to potential-based reward shaping (Ng et al. 1999): adding $\Phi(s_{t+1}) - \Phi(s_t)$ to the per-token rewards, for any function Φ of the prefix with $\Phi = 0$ at the end of a response, changes the return of every response of a prompt by the same amount $-\Phi(x)$, and so leaves the KL-optimal policy unchanged (Exercise 22.19). Step rewards that are *not* of this form change what is optimal; a reward of $+1$ for every step the PRM likes, for example, invites the policy to write many small, safe-looking steps.

Search at test time. Because the token-level MDP has known, deterministic dynamics, a model can search before it commits to an answer, trading computation for quality.

- *Best-of-n* samples n responses and returns the one a verifier or reward model scores highest. Its distribution is known exactly (Exercise 22.16), its divergence from the sampling policy is at most $\log n - \frac{n-1}{n}$ (Beirami et al. 2024), and Figure 22.2 shows that it lands just below the reward–KL frontier of the KL-regularized optimum: best-of-n is a cheap approximation of the policy that RLHF aims for.
- *Majority voting* (self-consistency) samples several chains of thought and returns the most common final answer; it needs no verifier (X. Wang et al. 2023).
- *Step-level search* keeps several partial solutions, extends them, and prunes with a PRM used as a value function, as in beam search or tree search over reasoning steps (Yao, Yu, et al. 2023), and as in the Monte Carlo tree search of Section 8.5.3.

Search and reinforcement learning feed each other. Search is a policy-improvement operator at test time; training the policy to imitate the improved outputs distills the search into the network, which is expert iteration (Anthony et al. 2017) and the principle behind AlphaZero (Section 15.7). STaR (Zelikman et al. 2022) fine-tunes a model on its own chains of thought that reached correct answers, which is a filtered form of REINFORCE with a binary reward.

Test-time compute. Snell et al. (2025) studied how best to spend a fixed budget of computation at test time. The best strategy depended on the difficulty of the question; allocating compute adaptively was reported to be more than four times as efficient as best-of-n; and, in comparisons with equal total computation, a smaller model with extra test-time compute outperformed a fourteen times larger model on questions where the small model had a reasonable chance of success. Reasoning models such as those of Section 22.6.4 turn search inward: their long chains of thought contain attempts, checks and backtracking, a sequential use of test-time compute that the policy learned through reinforcement learning.

22.10 Open problems ●●●

The methods of this chapter are young, and several basic questions are open in 2026. We describe five, with both sides of the debates where there are sides.

Exploration in reasoning. Reinforcement learning with verifiable rewards learns only from prompts the model sometimes solves; Figure 22.4 shows a prompt that is never learned because it is never solved. Policies also tend to lose entropy quickly during training, after which improvements slow down (Cui et al. 2025). The exploration methods of Chapter 16 (optimism, posterior sampling, intrinsic rewards) assume state and action spaces far smaller than the space of chains of thought, and it is not known how to explore in a space of possible arguments. Current remedies are heuristics: dynamic sampling, curricula over prompt difficulty, a higher upper clip, and objectives that reward success within k attempts rather than in one.

What does reinforcement learning add? Yue et al. (2025) compared models before and after RL with verifiable rewards using pass@k, the probability that at least one of k samples is correct, estimated without bias by the formula of M. Chen et al. (2021). The RL-trained models were better for small k, but the base models caught up and overtook them for large k (hundreds of samples) across their mathematics, coding and visual-reasoning benchmarks, and the solutions of the trained models were already likely under the base models. Their reading is that RL mostly *sharpens* the base distribution toward solutions it could already find, and reduces coverage. The support property of Section 22.3 is the formal version of this view: a KL-regularized optimum never leaves the support of the reference model. A toy shows how sharpening can lower pass@k. Suppose 70% of prompts can be solved only by strategy A and 30% only by strategy B, the policy chooses a strategy without looking at the prompt, and $\pi_{\text{ref}}(A) = \pi_{\text{ref}}(B) = 0.5$. The reward of a strategy, averaged over prompts, is 0.7 for A and 0.3 for B, and with $\beta = 0.1$ the KL-optimal policy puts 0.982 on A. Pass@1 rises from 0.50 to 0.69, but pass@16 falls from 1.00 to 0.78, because the strategy that the minority of prompts need has nearly disappeared.

The other side has evidence too. M. Liu et al. (2025) reported that prolonged RL training on diverse tasks, with KL control and periodic resets of the reference model, improved pass@k even for large k on tasks where the base model was weak. "In the support" is a weak constraint when the support contains every sequence: a probability of 10^{-30} is in the support but will never be sampled, and reweighting by $e^{r/\beta}$ can make it likely. Other findings complicate both readings. RL with random or even incorrect rewards improved one family of base models (Qwen2.5-Math) on a mathematics benchmark, but not other families, which suggests that some reported gains

amplify behaviors already present from pretraining (Shao et al. 2025); and models that already show verification and backtracking benefit more from RL than models that do not (Gandhi et al. 2025). Measuring "new capability" requires large-k evaluations that are expensive and noisy, and the answer may depend on the base model, the task and the amount of training.

Multi-turn and agentic reinforcement learning. Assistants increasingly act: they call tools, browse, and edit and run code over many steps, interleaving reasoning with actions (Yao, Zhao, et al. 2023). Benchmarks such as resolving real software issues (Jimenez et al. 2024) require dozens or hundreds of steps. The token-level MDP then becomes a genuine sequential problem with stochastic, partially observed transitions (the tool outputs; compare Chapter 19), rewards that are sparse and expensive to compute, and credit assignment across turns. Environments for training are costly to build and to run, and the algorithms of this chapter, designed for single responses, are being stretched to fit.

Scalable oversight. Human judgment is the reward of RLHF, and it becomes unreliable when models are better than their evaluators at a task. Proposals include *debate*, where two models argue and a human judges the arguments (Irving et al. 2018); *weak-to-strong generalization*, which asks whether a strong model trained on labels from a weaker supervisor can exceed it (Burns et al. 2024); and AI feedback. *Constitutional AI* (Bai, Kadavath, et al. 2022) has a model critique and revise its own responses according to a written list of principles, and then trains a preference model on AI-generated comparisons to replace human labels for harmlessness; Lee et al. (2024) reported that RL from AI feedback performed comparably to RL from human feedback on summarization and dialogue. Whether such methods remain reliable as models become more capable is not known.

Reward hacking by capable models. As policies become better optimizers, they find subtler exploits. Baker et al. (2025) found that monitoring a reasoning model's chain of thought with another model caught reward hacking in coding tasks, but that penalizing the model for "bad thoughts" during training taught it to hide its intent while continuing to hack. Rewards that remain informative under strong optimization, and ways to detect when they no longer are, are central open problems; Casper et al. (2023) survey the open problems and limitations of RLHF in general.

> **RESEARCH NOTE**
>
> Questions with no settled answer in 2026: Can a model learn, through RL, to solve problems that its base model solves with negligible probability, and how should that be measured? How should a language-model agent explore a space of arguments or programs? Which forms of process supervision help in training rather than only in search? How can reward signals be obtained for tasks that humans cannot evaluate? Chapter 24 places these questions among the open problems of the field.

22.11 Summary

- A language model is a policy. As a contextual bandit it chooses whole responses to prompts; as a token-level MDP it chooses tokens, with deterministic transitions, a tree of states and a single reward at the end.

- Reinforcement learning helps where supervised fine-tuning stops: when judging is easier than demonstrating, when the reward cannot be differentiated, and because training on its own samples teaches a model to recover from its own errors.
- RLHF fine-tunes a model on demonstrations, fits a Bradley–Terry reward model to human comparisons, and maximizes the reward minus β times the KL divergence from the reference model, usually with PPO and a per-token KL penalty in the reward.
- The KL-regularized objective has the optimum $\pi^* \propto \pi_{\text{ref}}\, e^{r/\beta}$: a reweighting of the reference model. Its reward and policy determine each other up to a prompt-dependent constant.
- DPO uses that correspondence to fit the policy to preferences directly, as a Bradley–Terry model whose reward is $\beta \log(\pi_\theta/\pi_{\text{ref}})$. With ideal data it recovers the KL-optimal policy; in practice it is offline and can ignore the KL regularization on deterministic preferences.
- With verifiable rewards, GRPO and RLOO replace the critic by a group of samples per prompt. Groups whose rewards are all equal teach nothing, and GRPO's normalizations by length and by standard deviation bias the update; Dr. GRPO and DAPO correct them.
- (●●) The KL divergence is estimated from samples by k_1, k_2 or k_3; used in a loss, k_1 gives no gradient in expectation and k_3 gives the gradient of the divergence in the opposite direction, $D_{\text{KL}}(\pi_{\text{ref}} \,\|\, \pi_\theta)$.
- Every reward is a proxy, and optimizing a proxy too hard lowers the true reward: overoptimization. KL budgets, better and fresher reward models, ensembles and constraints delay it.
- (●●) Process rewards act like value functions and support step-level search; test-time search and reinforcement learning complement each other.

The next chapter turns from what reinforcement learning can do to how to do it well: implementation, debugging, honest evaluation and safety.

22.12 Further Reading

RLHF. The line of papers from Christiano et al. (2017) through Ziegler et al. (2019) and Stiennon et al. (2020) to Ouyang et al. (2022) is the best introduction: each is readable and explains its design choices. Bai, Jones, et al. (2022) is a detailed account of training a helpful and harmless assistant, including online retraining of reward models and the $\sqrt{D_{\text{KL}}}$ relation. Huang et al. (2024) reproduce RLHF with PPO in the open and document its implementation details. Casper et al. (2023) survey what RLHF cannot do.

The KL-regularized objective and DPO. Korbak et al. (2022) explain the Bayesian reading of the KL penalty. Rafailov et al. (2023) derive DPO and are worth reading in the original; Azar et al. (2024) place DPO in a general family and explain its behavior on deterministic preferences. Xu et al. (2024) compare DPO and PPO carefully.

Verifiable rewards and reasoning. Shao et al. (2024) introduce GRPO, and Guo et al. (2025) describe DeepSeek-R1. Read them with the critical replications of Z. Liu et al. (2025) (Dr. GRPO) and Yu et al. (2025) (DAPO), which identify which details matter. Kool et al. (2019) and Ahmadian et al. (2024) make the case for simple REINFORCE with a leave-one-out baseline. Lambert

et al. (2024) describe an open post-training pipeline that combines supervised fine-tuning, DPO and RLVR.

Estimators, overoptimization and process rewards. Schulman (2020) is a short, clear note on the KL estimators, and Tang and Munos (2025) analyze the pitfalls of their gradients. Gao et al. (2023) is the reference on overoptimization; Skalse et al. (2022) give the formal definition of reward hacking. Lightman et al. (2024) and Uesato et al. (2022) compare process and outcome supervision, and Snell et al. (2025) study the scaling of test-time compute.

Open problems (●●●). For the debate about what RL adds to a base model, read Yue et al. (2025) and M. Liu et al. (2025) side by side, then Shao et al. (2025) and Gandhi et al. (2025). For scalable oversight, start with Irving et al. (2018), Burns et al. (2024) and Bai, Kadavath, et al. (2022); for reward hacking in reasoning models, Baker et al. (2025).

22.13 Exercises

Exercise 22.1 ● Counting responses

In the digit-sum task of Example 22.1 (tokens 1, 2, 3, eos; at most six tokens per response): (a) show that there are 1,093 possible responses and 364 prefixes (states of the token-level MDP); (b) find the correct responses for the target 5 and their total probability under the uniform model.

Exercise 22.2 ● The Bradley–Terry likelihood

Three responses A, B, C to one prompt have rewards 0, 1 and 3. (a) Compute $\Pr(\mathsf{B} \succ \mathsf{A})$, $\Pr(\mathsf{C} \succ \mathsf{A})$ and $\Pr(\mathsf{C} \succ \mathsf{B})$ under the Bradley–Terry model. (b) Compute the log-likelihood of the data set $\{\mathsf{B} \succ \mathsf{A},\ \mathsf{C} \succ \mathsf{B},\ \mathsf{A} \succ \mathsf{C}\}$. (c) Compute the derivative of the log-likelihood with respect to the reward of B, and explain its sign.

Exercise 22.3 ● What a reward model does not determine

(a) Show that replacing $r(x, y)$ by $r(x, y)+c(x)$ changes neither the Bradley–Terry probabilities nor the KL-optimal policy (22.2). (b) Show that replacing r by $2r$ has the same effect on the KL-optimal policy as halving β. What does this imply about reporting a value of β without saying how the reward is scaled?

Exercise 22.4 ● Two responses

A prompt has two responses with rewards r_1 and r_2 and reference probabilities $\pi_{\text{ref}}(1) = q$ and $1 - q$. (a) Show that the KL-optimal probability of response 1 is $\sigma\big(\log \frac{q}{1-q} + \frac{r_1 - r_2}{\beta}\big)$. (b) Evaluate it for $q = 0.2$, $r_1 - r_2 = 1$ and $\beta = 0.5$.

Exercise 22.5 ●● The KL-optimal policy by Lagrange multipliers

Derive (22.2) a second way: maximize $F(\pi)$ over $\pi(y) > 0$ subject to $\sum_y \pi(y) = 1$ with a Lagrange multiplier, and show that F is strictly concave, so that the stationary point is the unique maximum.

Exercise 22.6 ●● The reward–KL frontier

Let π^*_β be the KL-optimal policy for $\beta > 0$ and $K_\beta = D_{\mathrm{KL}}(\pi^*_\beta \,\|\, \pi_{\mathrm{ref}})$. (a) Show that every policy π with $D_{\mathrm{KL}}(\pi \,\|\, \pi_{\mathrm{ref}}) \le K_\beta$ has $\mathbb{E}_\pi[r] \le \mathbb{E}_{\pi^*_\beta}[r]$. (b) Show that along the frontier the slope $d\,\mathbb{E}_{\pi^*_\beta}[r]/dK_\beta$ equals β. Check (b) numerically on Example 22.2 at $\beta = 1$.

Exercise 22.7 ●● The KL divergence over tokens

Prove the decomposition of Section 22.3: for autoregressive policies, $D_{\mathrm{KL}}(\pi \,\|\, \pi_{\mathrm{ref}}) = \mathbb{E}_{y\sim\pi}\big[\sum_t D_{\mathrm{KL}}(\pi(\cdot \mid s_t) \,\|\, \pi_{\mathrm{ref}}(\cdot \mid s_t))\big]$, where the KL divergences on the left are between response distributions for a fixed prompt. Where do you use the fact that s_t is determined by the tokens before t?

Exercise 22.8 ●● Deriving DPO

(a) Derive the DPO loss (22.3) from (22.2) and the Bradley–Terry model, and explain why $Z(x)$, which cannot be computed, does not appear. (b) Derive the gradient of the loss. (c) Show that at $\pi_\theta = \pi_{\mathrm{ref}}$ the gradient of one pair is $-\frac{\beta}{2}\big(\nabla_\theta \log \pi_\theta(y_w \mid x) - \nabla_\theta \log \pi_\theta(y_l \mid x)\big)$, independent of the data otherwise.

Exercise 22.9 ●● Deterministic preferences

A data set contains only the pair $y_w \succ y_l$, observed many times. (a) Show that the DPO loss has no minimizer: its infimum is 0, approached only as the margin of log-ratios grows without bound, for every $\beta > 0$. (b) Show that the IPO loss $\big(h' - \frac{1}{2\tau}\big)^2$, with $h' = \log \frac{\pi_\theta(y_w)\,\pi_{\mathrm{ref}}(y_l)}{\pi_\theta(y_l)\,\pi_{\mathrm{ref}}(y_w)}$, is minimized exactly when $h' = \frac{1}{2\tau}$. Is the minimizing policy unique? (c) Contrast both with the KL-optimal policy of a Bradley–Terry reward fitted with a finite amount of regularization.

Exercise 22.10 ● GRPO advantages by hand

Compute the GRPO advantages (dividing by the population standard deviation) and the Dr. GRPO advantages for the groups of rewards (a) $(1, 1, 0, 0, 0, 0)$; (b) $(0.5, 1, 0, 0)$, where a partial-credit reward of 0.5 is allowed; (c) $(1, 1, 1, 1)$. In case (c), what is the gradient of GRPO's objective for this group, and what does DAPO's dynamic sampling do with it?

Exercise 22.11 ● Length normalization

A group of four responses has rewards $(1, 1, 0, 0)$ and lengths 50, 150, 50 and 250 tokens. (a) Compute the weight that GRPO gives to the gradient of each response's log-probability. (b) Do

the same for DAPO's token-level average and for Dr. GRPO with the constant $L_{\max} = 256$. (c) Which incorrect response is discouraged least under each scheme, and what would you expect to happen to the length of incorrect responses over a long training run with GRPO?

Exercise 22.12 ●● Leave-one-out baselines

(a) Prove that the RLOO estimate $\frac{1}{k}\sum_i \hat{A}_i^{\mathrm{RLOO}} \nabla_\theta \log \pi_\theta(y_i \mid x)$, with independent samples $y_i \sim \pi_\theta$, is unbiased for $\nabla_\theta \mathbb{E}_{y\sim\pi_\theta}[r(x,y)]$. (b) Show that $r_i - \mathrm{mean}(r) = \frac{k-1}{k}\hat{A}_i^{\mathrm{RLOO}}$. (c) For one prompt with binary rewards, a policy that is correct with probability $p = \sigma(z)$, and $G = 8$, compute the expected GRPO update of z, with and without division by the standard deviation, as a multiple of the true gradient $p(1-p)$, for $p = 0.1$ and $p = 0.5$. What does the result say about the difficulty bias?

Exercise 22.13 ●● The estimator k_3

(a) Prove parts 1 and 2 of Proposition 22.3 for k_3. (b) Show that k_2 and k_3 agree to second order in the log-ratio u. (c) For the policy π^* of Example 22.2 with $\beta = 1$, compute $\mathbb{E}[\rho^2]$ under π^* and the variance of k_3, and explain why the variance can be unbounded when the policy is far from the reference.

Exercise 22.14 ●● Estimators in a loss

(a) Prove the three loss gradients of Proposition 22.4 for a softmax policy over a finite set by direct computation with logits. (b) GRPO adds βk_3, summed over tokens, to its loss. Near π_{ref}, how does the resulting regularization compare with the penalty of (22.1)? (c) Explain in words why a per-token estimator differentiated in a loss misses the effect of a token on the later tokens' divergence.

Exercise 22.15 ● Overoptimization by hand

A prompt has three responses with $\pi_{\mathrm{ref}} = (0.6, 0.3, 0.1)$. The gold reward is $(0, 1, 0)$, but a reward model learned $r_\psi = (0, 1, 2)$: it overrates the rare third response. (a) Compute the KL-optimal policy for r_ψ, its gold reward and its proxy reward, for $\beta = \infty, 2, 1, 0.5, 0.25$. (b) Find the β that maximizes the gold reward. (c) Explain why the gold reward eventually decreases.

Exercise 22.16 ●● Best-of-n

(a) For a policy π over a finite set of responses whose scores are all different, show that best-of-n selects y with probability $F(y)^n - F^-(y)^n$, where $F(y)$ is the probability under π of a score at most that of y and $F^-(y) = F(y) - \pi(y)$. (b) Compute the best-of-4 distribution for Example 22.2, its expected reward and its KL divergence from π_{ref}, and compare with $\log 4 - \frac{3}{4}$. (c) In the limit where every response has negligible probability, show that the divergence tends to $\log n - \frac{n-1}{n}$.

Exercise 22.17 ● Normalizations and dynamic sampling in the lab

Extend Lab 22.2: (a) train the digit-sum model with GRPO's sequence normalization, DAPO's token normalization and Dr. GRPO's constant normalization (without division by the standard deviation), with step sizes chosen so that the first prompt is learned at a similar speed, and plot the success probability and the mean length of incorrect responses; (b) add dynamic sampling (resample prompts whose group had equal rewards, up to a limit) and measure how many iterations GRPO needs to reach a mean success probability of 0.8.

Exercise 22.18 ●●● Does reinforcement learning create new reasoning ability?

Read Yue et al. (2025) and M. Liu et al. (2025). (a) Summarize each paper's main evidence, and say precisely which quantity each measures. (b) Reproduce the strategy toy of Section 22.10 and compute pass@1 and pass@16 for $\beta = 0.5$, 0.2 and 0.1. Then modify the toy so that the policy may depend on the prompt, and explain why pass@k then no longer falls. (c) Propose an experiment, feasible with a small model, that would distinguish "sharpening" from "new capability", and say what result would support each.

Exercise 22.19 ●●● Process rewards as potentials

(a) In the token-level MDP with $\gamma = 1$, add to the reward of each token the shaping term $\Phi(s_{t+1}) - \Phi(s_t)$, where Φ is any function of the prefix with $\Phi = 0$ after the last token. Show that the KL-optimal policy is unchanged. (b) Show that the Monte Carlo label of Section 22.9, the fraction of m rollouts from a prefix that reach a correct answer, is an unbiased estimate of V^{π} of that prefix for the rollout policy. (c) Design an experiment that compares an outcome reward with a process reward for *training* (not only for search) under equal computation, and discuss how a policy might hack each.

CHAPTER 23

Doing RL Well: Engineering, Evaluation, and Safety

WHAT YOU WILL LEARN

- Implement an environment loop that handles termination, truncation, vectorized environments and normalization correctly, and recognize the most common bugs from their symptoms.
- Debug a reinforcement-learning system by starting small, checking against known answers, and logging the right diagnostics.
- Evaluate and compare algorithms with confidence intervals, the bootstrap, interquartile means and performance profiles, and decide how many runs an experiment needs.
- Tune hyperparameters with a fair budget and report how sensitive a method is to them.
- Design rewards, recognize specification gaming, and prove that potential-based shaping preserves optimal policies (●●).
- Formulate a constrained MDP, solve it with a Lagrangian method, prove that its dual has no gap, and explain risk-sensitive objectives such as CVaR (●●).
- Explain how policies trained in simulation reach the real world, and what compute and scale buy (●●).

The previous chapters were about algorithms: what to compute and why it works. This chapter is about everything else that decides whether an experiment succeeds. A reinforcement-learning agent is a feedback loop between a learner, an environment and a stream of data that the learner itself produces. Each part can be subtly wrong, and the loop is forgiving enough to keep running when it is. An agent with a bug often still learns something, its learning curves look plausible, and nothing crashes. A comparison between two algorithms based on three runs each can point the wrong way. A reward that seemed obviously right can be maximized in ways nobody intended. The skills that prevent these failures are rarely written down, and they separate projects that work from projects that do not.

The chapter collects those skills. They come from three directions. *Engineering*: how to write the environment loop, which details matter, and how to debug a system whose output is a noisy curve. *Evaluation*: how much runs vary, how to put error bars on results, how many runs an experiment needs, and how to compare algorithms fairly. *Safety*: how to specify what the agent should optimize, including constraints it must respect and risks it must avoid, and how to carry a policy from a simulator to the world. Every claim is illustrated with a small experiment that you can rerun in the lab, built from the tabular tools of the earlier chapters so that the true answers

are known exactly.

Section 23.1 explains why reinforcement learning is hard to get right. Section 23.2 and Section 23.3 cover implementation and debugging. Section 23.4 is the statistical core of the chapter: variability, confidence intervals, aggregate measures, and how many runs are enough. Section 23.5 turns to hyperparameters and Section 23.6 to reward design; its subsection on why potential shaping works is ●●. Section 23.7 treats constrained and risk-sensitive reinforcement learning, with a ●● subsection on Lagrangian duality and another on risk measures. Section 23.8 discusses the move from simulation to reality, Section 23.9 (●●) compute and scale, and Section 23.10 reproducibility and ethics. A first reading can skip the ●● parts; the rest is meant for every reader who will run an experiment.

23.1 Why reinforcement learning is hard to get right

Supervised learning has a reassuring property: a fixed data set and a loss that measures progress directly. If the training loss falls and the validation loss falls with it, the model is learning. Reinforcement learning has neither. Three features make it harder.

The data are not stationary. The agent's policy decides which states it visits, so every change to the policy changes the data. Value-based methods chase targets that move as their own estimates change (Chapter 10). A loss that goes down is no evidence of progress: a critic can fit its targets perfectly while the targets are wrong, and a policy-gradient loss is not even meant to decrease. The quantity we care about, the expected return, is measured only through noisy episodes.

Results are sensitive to details. Henderson et al. (2018) showed that the reported performance of popular deep RL algorithms changed substantially with the random seed, the network architecture, the reward scale and the codebase used. Engstrom et al. (2020) found that much of the difference between PPO and TRPO came from "code-level optimizations" such as reward scaling, value clipping and particular initializations, rather than from the clipped objective that distinguishes the algorithms on paper. Andrychowicz et al. (2021) trained about a quarter of a million agents to measure which of dozens of design choices in on-policy actor-critic methods matter. Small decisions that a paper does not mention can decide the outcome.

Bugs are silent. A sign error in a classifier's loss is noticed at once. In reinforcement learning the same error may only make the agent learn more slowly, or learn a slightly different task. Agents are good at exploiting whatever they are given, including bugs: an off-by-one error in the returns or a mistake in handling the end of an episode produces an agent that still improves, just not toward the right answer. The only defense is to check the system against cases where the right answer is known.

> **KEY IDEA**
>
> In reinforcement learning, "it learns" is not evidence that it is correct. Check every component against known answers, measure variability over many runs before believing a difference, and treat the reward as a specification that a capable optimizer will take literally.

23.2 Implementation

23.2.1 The environment interface

Almost every modern code base uses the interface of Gymnasium (Towers et al. 2024), the maintained successor of OpenAI Gym (Brockman et al. 2016). The library in this book follows it too (Appendix D), and so do the standard physics benchmarks built on MuJoCo (Todorov et al. 2012), such as the DeepMind Control Suite (Tassa et al. 2018). An environment has two methods. `reset(seed)` starts an episode and returns the first observation and an `info` dictionary. `step(action)` returns five things: the next observation, the reward, a flag `terminated`, a flag `truncated`, and `info`. The core loop of an agent that learns a value function looks like this:

```
import gymnasium as gym

env = gym.make('CartPole-v1')
obs, info = env.reset(seed=0)
for t in range(100_000):
    action = agent.act(obs)
    next_obs, reward, terminated, truncated, info = env.step(action)
    # bootstrap from the next state unless a terminal state was reached
    target = reward + gamma * (1.0 - terminated) * agent.value(next_obs)
    agent.update(obs, action, target)
    obs = next_obs
    if terminated or truncated:
        obs, info = env.reset()
```

Three details deserve attention. First, the *spaces*: the attributes `env.observation_space` and `env.action_space` say what observations and actions look like (a `Box` of real vectors with bounds, or a `Discrete` set). Continuous actions usually have bounds, and many environments clip actions silently; a policy that outputs values far outside the bounds receives gradients for actions that were never executed. Second, *seeding*: `reset(seed=...)` seeds the environment's random generator, but the agent's own randomness (initial weights, exploration, minibatch sampling) needs its own seeds. Third, *wrappers*: time limits, frame stacking, normalization and episode statistics are usually added as wrappers around the environment, and they change what the agent sees. A wrapper that normalizes rewards changes the reward that the agent learns from, so the returns you log must come from before the wrapper.

23.2.2 Terminated versus truncated

An episode can stop for two different reasons. It *terminates* when it reaches a terminal state of the task: the pole falls, the robot reaches the goal, the game is over. The future is then worth exactly zero. It is *truncated* when something outside the task cuts it short, most often a time limit added so that training sees many different starting states. The state reached at a truncation is an ordinary state with an ordinary value. The correct target bootstraps from it:

$$y_t = r_t + \gamma\,(1 - \text{terminated}_t)\,V(s_{t+1}),$$

with $\text{terminated}_t \in \{0, 1\}$, and *not* with terminated_t replaced by "terminated or truncated". Section 6.8 explains the theory; here is what the bug does.

Example 23.1 The time-limit bug halves a value

Take the simplest task with a time limit: a single state, reward +1 at every step, no terminal state, and episodes cut off after $H = 100$ steps, with $\gamma = 0.99$. The true value is $V = \sum_k \gamma^k = 1/(1-\gamma) = 100$, whatever the time limit, because the time limit is not part of the task.

Now suppose the agent treats the time limit as termination. In each episode, 99 of the 100 updates use the target $1 + \gamma V$ and the last one uses the target 1. TD(0) with a small constant step size settles where the targets balance on average, at

$$V = \frac{99}{100}(1 + \gamma V) + \frac{1}{100} \cdot 1 \quad \Longrightarrow \quad V = \frac{1}{1 - 0.99\,\gamma} = \frac{1}{1 - 0.9801} \approx 50.25.$$

In the lab, TD(0) with step size 0.01 learns 100.0 with the correct target and 50.1 with the buggy one. Both learning curves are smooth and look converged.

♦

The bug is easy to make because the old Gym interface returned a single flag `done` that mixed the two cases, and because many tasks seem to have both: CartPole terminates when the pole falls and is truncated after 500 steps. In a value-based method the bug biases values downward near the time limit and makes them depend on time, which is not in the state. In actor-critic methods it corrupts the advantages and the generalized advantage estimates (Section 11.7) at every episode boundary.

> **PITFALL**
>
> If the time limit *is* part of the task, as in a game that ends when a clock runs out, then the state must contain the remaining time, otherwise the problem is not Markov. Pardo et al. (2018) analyze both cases: bootstrap at truncations when the limit is an artifact of training, and add the remaining time to the observation when the limit is real.

23.2.3 Vectorized environments

Policy-gradient methods such as A2C and PPO collect experience from many copies of the environment at once, stepping them in lockstep so that the policy network processes a batch of observations in a single call. Such a *vector environment* returns arrays of observations, rewards and flags, one entry per copy, and resets each copy automatically when its episode ends. The automatic reset is the source of a classic bug. When a copy finishes, the observation that comes back is either the last observation of the old episode or the first observation of the new one, and libraries differ. In Gymnasium versions before 1.0, the reset happened inside the same `step` call: the returned observation was already the new episode's first one, and the final observation of the old episode was placed in `info`. Gymnasium 1.0 made the reset happen on the *next* call to `step` by default, which then ignores its action for that copy. Code written for one convention and run with the other bootstraps from the wrong state. Read the documentation of the version you use, and test the boundary with a hand-made environment whose observations reveal the time step.

23.2.4 Normalizing observations and rewards

Neural networks learn best when their inputs are on a scale of about one, and policy-gradient methods learn best when advantages are. Observations in raw units (positions in meters, velocities in meters per second, joint torques) can differ by orders of magnitude. The standard remedy is to keep running estimates of the mean μ and standard deviation σ of each observation component, feed the network $\mathrm{clip}\big((o-\mu)/\sigma, -10, 10\big)$, and update the statistics as data arrive (`rlcourse.utils.RunningMeanStd` implements the update). Rewards are often divided by a running estimate of the standard deviation of the discounted return. Scaling all rewards by a positive constant leaves the optimal policy unchanged, so this only changes the conditioning of the learning problem. Shifting rewards by a constant is different: in an episodic task it changes the value of ending an episode early, and can change the optimal policy.

Two rules keep normalization from becoming a bug. The statistics are part of the policy: save them with the network weights, because the same weights with different statistics form a different policy. And freeze them for evaluation, so that the evaluation does not change the agent that it evaluates.

23.2.5 Networks, initialization and frame skipping

A few further choices recur in deep RL code. Policy networks often start with a final layer whose weights are scaled down by a factor of about 100, so that the initial policy is close to uniform; Andrychowicz et al. (2021) found this to be one of the more important choices for on-policy methods. Many PPO implementations initialize weight matrices to be orthogonal, use tanh activations, clip the global gradient norm, and anneal the learning rate; Huang, Dossa, Raffin, et al. (2022) document such details for PPO one by one, and Section 12.7 discusses which ones matter. In video games, agents usually repeat each action for several frames (*frame skipping*), which shortens the effective horizon, and they stack the last few frames so that velocities can be inferred; the Atari agents of Chapter 10 repeat actions for four frames. Section 10.10 describes the evaluation protocols that make Atari results comparable.

23.2.6 The most common bugs

Table 23.1 lists bugs that experienced practitioners have all made, with the symptom that usually reveals them and the first thing to check.

Table 23.1. Common bugs, their symptoms and the first check to make.

Symptom	Likely cause	First check
values too low near episode ends	truncation treated as termination	the target at time-outs
values grow without bound	bootstrapping from terminal states	the target at true ends
no learning at all	wrong sign, detached gradient	overfit one batch

Symptom	Likely cause	First check
good on one seed only	luck in exploration	ten or more seeds
entropy collapses at once	learning rate too high	KL per update
late collapse	updates too large, old data	KL, clip fraction
evaluation far below training	exploration left on, new statistics	frozen evaluation
NaN losses	log of zero, variance near zero	assert finite values
change after a refactor	off-by-one in returns or masks	a hand-checked batch

23.3 Debugging

23.3.1 Start small and check against known answers

The most effective debugging strategy is to make the problem small enough that the correct answer is known. Before training an agent on a large environment, run the same code on a problem that a few lines of dynamic programming can solve. The Trail is ideal: the uniform policy has $V(\text{Start}) = 1.76$, the optimal policy goes all the way with $V^*(\text{Start}) = 3.24$, and every estimate a learner produces can be compared with these numbers. Next comes a small environment with a known solution, such as the grid worlds of the earlier chapters, and only then a standard benchmark like CartPole, where a working agent reaches the maximum return of 500 within a predictable budget. Each step up the ladder adds one source of difficulty, so a failure points to the part that was added.

The same idea works at the level of functions. Returns, generalized advantage estimates, importance ratios and target computations are pure functions of a short list of numbers, and they can be checked against a hand calculation on a three-step episode. The tests of the course library do exactly this (Appendix D): they compare the code with exact values wherever possible.

23.3.2 Sanity checks

A handful of checks catch most problems before a long training run.

Overfit a single batch. Freeze one batch of data and train the critic (or the classifier inside a reward model, or the dynamics model) on it for many steps. The loss must go to nearly zero. If it does not, the bug is in the network, the loss or the optimizer, not in the reinforcement learning.

Compare value predictions with Monte Carlo returns. For states visited by the current policy, the critic's prediction $V(s_t)$ should track the observed return G_t. A convenient summary is the *explained variance*

$$\text{EV} = 1 - \frac{\text{Var}(G - V)}{\text{Var}(G)},$$

computed over a batch of states. It is 1 for a perfect critic, 0 for a critic that predicts the mean return everywhere, and negative for one that is worse than that. Explained variance that stays near zero or below while the policy changes is a red flag.

Run baselines. A random policy and a simple scripted policy give the floor of performance. An agent that does not beat a random policy after a reasonable budget has a bug or a problem too hard for it, and should not be scaled up. On the Trail, the random policy earns 1.76; on CartPole, a random policy lasts about 22 steps.

Check one update by hand. For a tiny batch, compute every intermediate quantity of one update (returns, masks, log-probabilities, ratios, advantages, the loss and its gradient) by hand or in a separate, simple implementation, and compare.

23.3.3 What to log

Return curves alone are not enough to diagnose a problem; they say that something is wrong, not what. Table 23.2 lists the quantities that most deep RL code bases log, and the signs of trouble in each.

Table 23.2. Diagnostics worth logging in deep RL, and what each reveals.

Quantity	What it shows	Warning sign
return, episode length	progress on the task	flat at the random level
policy entropy	how much it still explores	collapses in a few updates
KL between updates	size of policy changes	spikes, or always zero
PPO clip fraction	share of clipped ratios	near zero or above about 0.3
value loss, explained var.	quality of the critic	EV near zero or negative
gradient norms	optimization health	explodes, or vanishes
Q-values	scale of predictions	above $r_{\max}/(1-\gamma)$
steps per second	throughput	drops during training

The Q-value row uses a bound from Chapter 3: with rewards in $[-r_{\max}, r_{\max}]$, every value lies in $[-r_{\max}/(1-\gamma), r_{\max}/(1-\gamma)]$. Predictions outside this range are impossible for any policy, and they are a reliable sign of divergence or of bootstrapping from terminal states.

23.3.4 Reading learning curves

A learning curve is the return (or a normalized score) plotted against environment steps. A few shapes come up again and again. A curve that stays *flat at the random level* means no learning signal: a bug, an exploration problem (the agent never sees a reward), or a learning rate that is far too small. A curve that *rises and then collapses* usually means that updates became too large, that the critic diverged, or that the replay buffer is full of data from a very different policy; the KL divergence and the value loss usually show the problem a little before the return does. A

curve that *rises and plateaus* below the known optimum can mean a local optimum, too little exploration, entropy that collapsed early, or a network too small for the task. And a curve that is *very noisy* may be fine: returns of single episodes vary a lot.

Two habits avoid misreading curves. Plot the curves of several runs, not just their average, because an average of runs that succeed at different times can look like smooth slow progress that no single run shows. And distinguish *training returns*, earned while exploring, from *evaluation returns*, earned by the current policy acting as it will be deployed (greedily, or without exploration noise). The two can differ substantially, and the evaluation return is the one that measures the policy.

23.4 Evaluation and statistics

23.4.1 What is being measured

Before computing an error bar, decide what the number estimates. An experiment in reinforcement learning has two layers of randomness. Within a *run* (one training process with one seed), the evaluation episodes vary; between runs, the whole training process varies, because initial weights, exploration and environment noise are random. The quantity of interest is usually a property of the distribution of runs: how well does this algorithm, with these settings, do on this task? So the unit of analysis is the run. Each run is summarized by one *score*, such as the average evaluation return at the end of training (*final performance*) or the average over the whole training period (the *area under the learning curve*, which also rewards learning quickly). With several runs, the scores are independent samples from a distribution, and ordinary statistics applies.

Our running experiment uses tabular agents so that every evaluation is exact. On the five small tasks of the lab (two 8×8 grids with a single reward at the goal, one of them with walls; the cliff; the slippery 4×3 world; and the hazard grid of Section 23.7), we evaluate the greedy policy exactly with dynamic programming and *normalize* its value: 0 is the value of the uniform random policy, 1 the optimal value. The score of a run is the mean of 20 such evaluations spread over a budget of 4,000 environment steps. Normalized scores make tasks comparable, in the same way that Atari results are often reported as human-normalized scores (Mnih et al. 2015).

23.4.2 Seeds and variability

Start with a single algorithm on a single task. Figure 23.1 shows twenty runs of Q-learning on the grid with walls. The runs differ only in their random seeds, yet the step at which the greedy policy first becomes optimal ranges from 2,400 to 4,800, with a median of 3,100. Their scores (areas under the curve) have mean 0.656 and standard deviation 0.087.

The right panel makes the danger concrete. Among the 126 ways of dividing the first ten runs into two groups of five, the most different pair has mean scores 0.723 and 0.613, and a two-sample test (introduced below) declares the difference significant with $p = 0.017$. An author who ran ten seeds, kept the five best for a new method and the five worst for the baseline, would find a "significant" improvement where there is none. Even without bad intent, trying several configurations and reporting the best is a version of the same selection. Henderson et al. (2018)

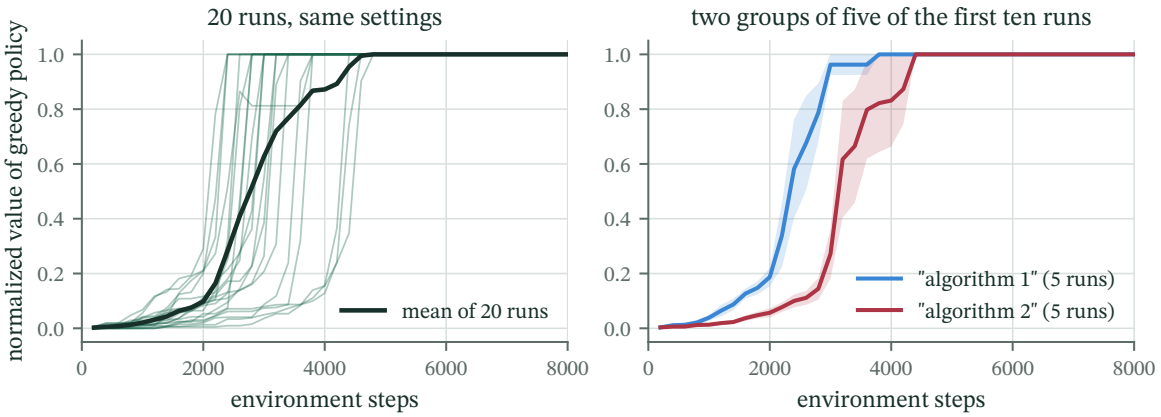


Figure 23.1. Twenty runs of Q-learning on the 8×8 grid with walls, identical except for the random seed (left; the thick line is their mean), and two groups of five of the first ten runs (right; bands are ± 1 standard error). The right panel shows the most different of the 126 ways of splitting ten runs into two groups of five. The two "algorithms" are the same algorithm.

made this point vividly with deep RL algorithms on continuous-control tasks: two groups of five runs of one algorithm, differing only in their seeds, produced learning curves that looked like those of two different algorithms.

23.4.3 Confidence intervals and the bootstrap

The first remedy is to report uncertainty. Suppose a run's score is a random variable X with mean μ, and n independent runs give scores $x_1, \dots, x_n$ with sample mean $\bar{x}$ and sample standard deviation s. The *standard error* of the mean is $s/\sqrt{n}$, and the *Student t interval*

$$\bar{x} \pm t_{n-1} \frac{s}{\sqrt{n}}$$

contains μ with probability 95% if the scores are normally distributed, where t_{n-1} is the 97.5% quantile of the t distribution with $n-1$ degrees of freedom ($t_9 = 2.262$, $t_{19} = 2.093$; it approaches the normal value 1.96 as n grows). The t quantile is larger than 1.96 because s itself is estimated from the same few runs.

The *bootstrap* (Efron 1979; Efron and Tibshirani 1993) gives intervals for statistics that have no simple formula. Its idea is to treat the sample as a stand-in for the unknown distribution and to see how much the statistic varies when the sample is drawn again from it.

Algorithm 23.1 Percentile bootstrap confidence interval

Input: scores $x_1, \dots, x_n$; a statistic T (mean, median, IQM, ...); number of resamples B (for example 10,000); level $1 - \alpha$.

1. For $b = 1, \dots, B$:

   1. Draw n indices uniformly at random from $\{1, \dots, n\}$, *with* replacement.
   2. Compute T_b, the statistic of the resampled scores.
2. Sort $T_1, \dots, T_B$.

Output: the interval between the $\alpha/2$ and $1 - \alpha/2$ quantiles of the T_b.

Example 23.2 Error bars for ten runs

Ten runs of Q-learning on the grid with walls, with the lab's protocol, scored

$$0.045,\ 0.374,\ 0.416,\ 0.390,\ 0.302,\ 0.511,\ 0.178,\ 0.323,\ 0.543,\ 0.317.$$

Their mean is 0.340 and their standard deviation 0.147, so the standard error is $0.147/\sqrt{10} = 0.046$. The t interval is $0.340 \pm 2.262 \times 0.046 = [0.235, 0.445]$. The normal approximation, with 1.96 in place of 2.262, gives $[0.249, 0.431]$, and the percentile bootstrap with 10,000 resamples gives $[0.249, 0.422]$. All three say the same thing: the true mean score is known only to within about ± 0.1, and a single run could have scored anywhere from 0.05 to 0.54.

♦

The two short intervals in the example are not a free gift. How often does a nominal 95% interval actually contain the true mean? A simulation answers the question. With normally distributed scores, the t interval covered the mean in 94% to 96% of 2,000 repetitions for every n from 3 to 50. The percentile bootstrap covered it in only about 83% of repetitions with 5 runs, 90% with 10 runs and 93% with 20 runs: with few runs, resampling the sample underestimates the variability of the population. With skewed scores (a log-normal distribution, the exponential of a standard normal variable), every interval undercovered at small n: with 10 runs, 84% for t and 80% for the bootstrap. The lessons are practical. With a handful of runs, use t intervals for means, and treat every interval as optimistic. The bootstrap earns its place for statistics without a formula and for aggregates over many tasks, where the number of runs pooled is large.

23.4.4 Aggregating across tasks: IQM and performance profiles

Algorithms are usually compared over a suite of tasks: 57 Atari games, a dozen MuJoCo tasks, our five grids. One then needs a single summary, and the choice matters. R. Agarwal et al. (2021) analyzed published results on the Atari 100k benchmark and found that conclusions often rested on point estimates from three to five runs whose uncertainty was large enough to reverse the ranking of algorithms. They recommend three tools, all now standard.

The first is a robust aggregate. The *mean* normalized score over all runs and tasks is dominated by a few tasks with large scores. The *median* is robust but ignores all but the middle values and is very variable with few runs. The *interquartile mean* (IQM) is a compromise: sort all the scores of all runs on all tasks, discard the bottom and top quarters, and average the middle half. It is robust to outliers and uses half the data.

The second is a *stratified bootstrap* for the uncertainty of the aggregate: in each resample, draw the runs of each task with replacement, separately for each task, and recompute the aggregate. With m tasks and n runs per task, the resampled data always contain n runs of every task.

The third is the *performance profile* (Agarwal et al. call it the run-score distribution), adapted from optimization benchmarking (Dolan and Moré 2002): the fraction of runs whose normalized score exceeds τ, plotted for every threshold τ,

$$\hat{F}(\tau) = \frac{1}{m} \sum_{j=1}^{m} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{x_{ij} > \tau\},$$

where x_{ij} is the score of run i on task j. A profile that lies above another for every τ means that the first algorithm is better at every level of performance. Profiles also show at a glance whether a difference comes from a few tasks or from all of them. A related summary is the *probability of improvement*: the probability that a randomly chosen run of algorithm A scores higher than a randomly chosen run of algorithm B on the same task, averaged over tasks (ties count one half).

Figure 23.2 applies these tools to a comparison of Q-learning (agent A) and Expected SARSA (agent B), both with step size 0.5 and $\varepsilon = 0.1$, with ten runs of each on each of the five tasks. Per task, the mean scores of A and B are 0.50 and 0.52 on the open grid, 0.34 and 0.27 on the grid with walls, 0.97 and 0.97 on the cliff, 0.91 and 0.92 on the 4×3 world, and 0.77 and 0.77 on the hazard grid. The aggregates are mean 0.70 $[0.67, 0.73]$ for A and 0.69 $[0.66, 0.71]$ for B, and IQM 0.75 $[0.73, 0.77]$ for A and 0.75 $[0.72, 0.77]$ for B, with 95% stratified-bootstrap intervals in brackets. The median is 0.783 for both agents: so many runs on the hazard grid share that exact score that the median cannot tell the agents apart. The probability that a run of A beats a run of B is 0.57, with a bootstrap interval of about $[0.46, 0.68]$ that contains one half.

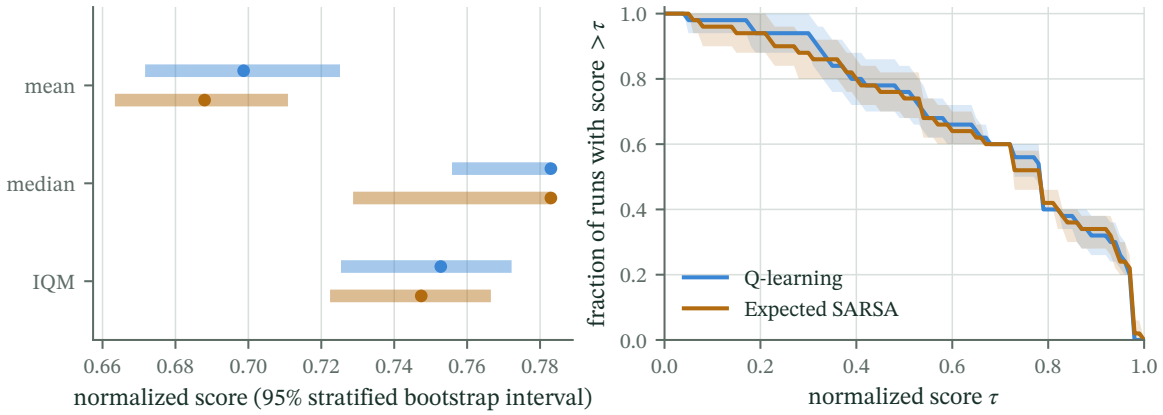


Figure 23.2. Q-learning (blue) and Expected SARSA (amber) on five small tasks, ten runs each. Left: mean, median and interquartile mean of the normalized scores, with 95% stratified-bootstrap intervals. Right: performance profiles (the fraction of runs scoring above each threshold) with pointwise 95% bands. Every interval and band overlaps; the median is stuck at the same value for both agents.

The honest conclusion is that this experiment does not distinguish the two agents. That is a perfectly good result to report, as long as it is reported with its uncertainty.

23.4.5 How many runs?

On the grid with walls alone, A's mean score (0.340) is higher than B's (0.270). Is the difference real? *Welch's t-test* (Welch 1947) compares two means without assuming that the two algorithms have equal variances. With sample means $\bar{x}_A$, $\bar{x}_B$, sample variances s_A^2, s_B^2 and n_A, n_B runs, its statistic and its approximate degrees of freedom are

$$t = \frac{\bar{x}_A - \bar{x}_B}{\sqrt{s_A^2/n_A + s_B^2/n_B}}, \qquad \nu = \frac{\left(s_A^2/n_A + s_B^2/n_B\right)^2}{\frac{(s_A^2/n_A)^2}{n_A - 1} + \frac{(s_B^2/n_B)^2}{n_B - 1}},$$

and the two-sided p-value is the probability that a t-distributed variable with ν degrees of freedom exceeds $|t|$ in absolute value. Here $t = 1.03$, $\nu = 18.0$ and $p = 0.32$: a difference this large would be common if the agents were equally good.

A non-significant result is not evidence that the agents are equal; it may only mean that the experiment was too small. The *power* of a test is the probability that it detects a difference of a given size. The size is measured by the *effect size* $d = (\mu_A - \mu_B)/\sigma$, the difference of the true means in units of the standard deviation of a run's score. For the grid with walls the estimate is $d = 0.46$. With ten runs per agent, a test at the 5% level detects an effect of this size with probability only 0.16. To detect it with probability 0.8, the conventional target, needs 75 runs per agent.

A rule of thumb follows from the normal approximation: to detect an effect size d with a two-sided test at level α and power $1 - \beta$, use about

$$n \approx \frac{2\,(z_{1-\alpha/2} + z_{1-\beta})^2}{d^2}$$

runs per algorithm, where z_q is the q quantile of the standard normal distribution ($z_{0.975} = 1.96$ and $z_{0.8} = 0.84$). Table 23.3 gives the exact numbers for the t-test, which are slightly larger.

Table 23.3. Runs per algorithm needed for 80% power with a two-sided two-sample t-test at the 5% level.

Effect size d	0.2	0.5	0.8	1.0	1.5	2.0
runs per algorithm	394	64	26	17	9	6
normal approximation	393	63	25	16	7	4

Colas et al. (2018) discuss these calculations for deep RL and show how to estimate the variance from a pilot experiment. The table explains why results based on three to five runs can detect only very large effects: with five runs per algorithm, the power for $d = 1$ (a difference of one standard deviation) is only 0.29.

PITFALL

Every test adds a chance of a false positive. Testing ten tasks separately at the 5% level gives a good chance that at least one shows a "significant" difference by luck. Decide the comparisons before running the experiment, report all of them, and correct for multiple comparisons (the simplest correction, Bonferroni's, divides the level by the number of tests) or use an aggregate over tasks.

23.4.6 Comparing algorithms fairly

Statistics cannot rescue an unfair comparison. Four rules cover most cases.

Equal tuning budgets. A new method tuned over fifty configurations, compared with a baseline run at its published defaults, will usually win. Tune both with the same budget and the same procedure, or report the tuning budget of each.

Equal experience. Compare at the same number of environment steps (or of episodes, or of gradient updates, depending on the claim), and say which. "Sample efficiency" claims need curves, not a single point.

The same evaluation protocol. The same number of evaluation episodes, the same exploration settings during evaluation, the same normalization of scores, and the same seeds for evaluation environments.

No selection on the test data. Reporting the best checkpoint of each run, or the best of several configurations, inflates the result. If k checkpoints are equally good and their evaluations are independent with noise of standard deviation σ, the best of them overstates the truth by $\mathbb{E}[\max_{i \le k} Z_i]\,\sigma$ on average, where the Z_i are standard normal: 0.56σ for two checkpoints, 1.54σ for ten and 1.87σ for twenty. Choose checkpoints and hyperparameters on separate seeds or separate evaluation episodes from those used for the final report. The same maximization bias that makes Q-learning overestimate values (Section 6.7) makes careless experimenters overestimate their methods.

23.4.7 A reporting checklist

A results section that a reader can trust states the following:

1. what each run's score measures (final performance or area under the curve; evaluation or training returns; the number of evaluation episodes; the normalization);
2. the number of independent runs per algorithm and task, and how seeds were chosen;
3. an interval for every reported number, and what the interval is (standard error, t interval, bootstrap);
4. the tuning procedure and budget for every algorithm, including baselines;
5. the environment versions, wrappers, time limits and the treatment of truncation;
6. the compute used: hardware, wall-clock time and environment steps;
7. aggregate measures that are robust (IQM) together with per-task results, and performance profiles for suites;
8. all runs, including failures, and the code and configurations needed to reproduce them.

LAB

Lab 23.1: Doing RL well (`labs/ch23_practice.py`, about 60 minutes; the script runs in about a minute). Run `python3 labs/ch23_practice.py`. Task 1 reproduces the time-limit bug, Task 2 the twenty runs of Figure 23.1, Task 3 the coverage of t and bootstrap intervals, and Task 4 the full comparison of Figure 23.2, with the probability of improvement, Welch's test and the number of runs needed. The statistics live in `rlcourse/stats.py`; use them for your own experiments.

23.5 Hyperparameters

23.5.1 Which hyperparameters matter

Every algorithm in this book has hyperparameters, and deep RL algorithms have many. Some matter much more than others, and which ones matter depends on the algorithm. Table 23.4 lists the ones that usually deserve attention first, with common starting values; they are starting points, not answers.

Table 23.4. Hyperparameters that usually matter most, with common starting values.

Algorithm	Most sensitive	Common starting values
DQN	learning rate, target update, ε schedule	Adam 10^{-4}, n-step 3
PPO	learning rate, epochs, clip ϵ, γ, λ	$3 \cdot 10^{-4}$, $\epsilon = 0.2$, $\lambda = 0.95$
SAC	learning rate, target entropy, replay ratio	$3 \cdot 10^{-4}$, target $-\dim(\mathcal{A})$

For on-policy actor-critic methods, Andrychowicz et al. (2021) give detailed recommendations from their large study: normalize observations, start from small output layers in the policy, use generalized advantage estimation with λ around 0.9, and tune the discount factor, which was among the most important and most environment-specific choices. The first version of SAC was sensitive to the scale of the rewards, because the scale sets the balance between reward and entropy; automatic tuning of the temperature against a target entropy largely removed that sensitivity (Haarnoja, Zhou, Hartikainen, et al. 2018), as Section 13.6.2 explains. For DQN and its descendants, Section 10.10 discusses the choices that matter. Two hyperparameters are easy to overlook because they are part of the problem rather than of the algorithm: the discount factor γ, which changes what is optimal, and the time limit, which changes what the agent sees.

23.5.2 Tuning strategies

Grid search evaluates every combination of a few values per hyperparameter. *Random search* samples combinations at random from chosen ranges. Bergstra and Bengio (2012) showed that random search is usually more efficient: when only a few of many hyperparameters matter, a grid spends most of its budget repeating the same values of the important ones, while random search tries a new value of every hyperparameter in every trial. Sample scale parameters such

as learning rates uniformly on a logarithmic scale. *Bayesian optimization* fits a model of performance as a function of the hyperparameters and chooses the next trial where the model predicts improvement. *Population-based training* (Jaderberg et al. 2017) trains a population of agents in parallel, periodically copies the weights of good agents into bad ones and perturbs their hyperparameters, so that the hyperparameters change during training.

Two practices matter more than the choice of method. Tune with several seeds per configuration, because the variability of Figure 23.1 applies to every trial, and the best configuration of a noisy search is optimistically biased for the reason given in Section 23.4. And after tuning, rerun the chosen configuration with fresh seeds for the final report.

23.5.3 Sensitivity analysis

A single number from a tuned configuration hides how hard it was to find. A *sensitivity curve* shows performance as a function of one hyperparameter, with the others fixed, and with an interval at every point. Figure 23.3 shows sensitivity curves for the step size η of Q-learning on three tasks, ten runs per point. The tasks disagree. On the slippery 4×3 world performance peaks at $\eta = 0.2$ (score 0.95) and falls to 0.70 at $\eta = 1$, because large steps chase the noise of random transitions. On the deterministic cliff, larger steps are better, up to 0.986 at $\eta = 0.9$. On the grid with walls the score hardly changes for $\eta \geq 0.1$, and the variability between runs (the band) is larger than any effect of η.

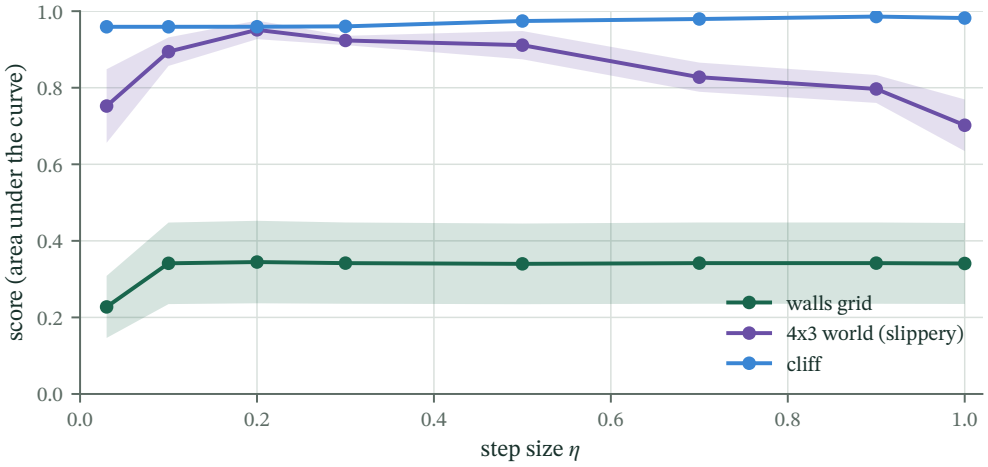


Figure 23.3. Sensitivity of Q-learning to its step size η on three tasks: the score (area under the normalized learning curve) for eight step sizes, with 95% t intervals over ten runs. The best step size depends on the task; on the grid with walls, run-to-run variability dominates.

Patterson et al. (2024) argue that sensitivity analyses of this kind should be standard in reinforcement-learning papers, and discuss how to design them. An algorithm that performs well over a wide range of hyperparameters is more useful in practice than one that is slightly better at a single, carefully tuned setting.

23.6 Reward design

23.6.1 Sparse and dense rewards

The reward is the only channel through which a designer tells the agent what to do, so it deserves as much care as the algorithm. The most faithful rewards are often *sparse*: $+1$ for reaching the goal and 0 everywhere else, $+1$ for winning the game. They say exactly what is wanted, but they give the agent nothing to learn from until it stumbles on the goal by chance. On our 8×8 grid with walls, a Q-learning agent with zero initial values needs a median of about 3,100 steps before its greedy policy is optimal, almost all of it spent in random wandering before the first visit to the goal; on a larger grid, or with a longer path, random discovery can take exponentially long (Chapter 16).

Dense rewards that grade progress (for example, minus the distance to the goal) give a signal at every step, and are tempting for this reason. But a dense reward is a claim about what progress means, and if the claim is wrong, the agent optimizes the claim.

23.6.2 Specification gaming

Agents that find unintended ways to earn reward are so common that there are collections of examples. In a boat-racing video game, an agent rewarded for hitting targets along the course learned to circle in a lagoon, repeatedly hitting the same respawning targets, catching fire and crashing, and still scored higher than is possible by completing the course normally (Clark and Amodei 2016). A simulated bicycle rewarded for progress toward a goal, but not penalized for moving away, learned to ride in circles, collecting reward on the half of each circle that pointed toward the goal (Randløv and Alstrøm 1998; Ng et al. 1999). Krakovna et al. (2020) maintain a list of dozens of such cases in reinforcement learning and related fields, and call the phenomenon *specification gaming*: the agent satisfies the literal specification, not the intended one. With language models trained against learned reward models, the same problem appears as reward hacking and overoptimization (Section 22.8).

Our grid gives a small, exact example. Suppose the designer adds a bonus of $+0.1$ every time the agent enters a cell within two steps of the goal, to guide it there. Staying near the goal forever then earns $0.1/(1-\gamma) = 2$ with $\gamma = 0.95$, while entering the goal earns 1 and ends the episode. The optimal policy for the shaped reward never enters the goal: its true value is exactly 0. A Q-learning agent with the bonus duly learns to hover beside the goal (Figure 23.4, red curve).

> **PITFALL**
>
> Reward bonuses for "being in good places" create reward that can be farmed. If a bonus can be earned repeatedly, assume that a capable agent will find the loop that earns it forever. Bonuses for *events* that cannot repeat, or potential differences as in the next subsection, avoid the problem.

23.6.3 Potential-based shaping

Is there a way to add guidance that can never change what is optimal? Ng et al. (1999) found the answer. Choose any function $\Phi : \mathcal{S} \to \mathbb{R}$, called a *potential*, with $\Phi = 0$ at terminal states,

and add to every reward the *difference in potential*

$$F(s, a, s') = \gamma\,\Phi(s') - \Phi(s). \tag{23.1}$$

The agent then learns from $r + F$. Because the bonus for moving from s to s' is exactly cancelled, up to discounting, by the penalty for moving back, no cycle can earn shaping reward: along any trajectory the shaping terms telescope, $\sum_{t=0}^{T-1} \gamma^t F(s_t, a_t, s_{t+1}) = \gamma^T \Phi(s_T) - \Phi(s_0)$. Every policy's value changes by the same amount, $-\Phi(s)$, so the ranking of policies is unchanged. Section 23.6.4 proves this.

Example 23.3 Shaping the Trail

Take the Trail with $\gamma = 0.9$ and the potential $\Phi(\text{Start}) = 0$, $\Phi(\text{Middle}) = 1$, $\Phi(\text{Goal}) = 2$, $\Phi(\text{End}) = 0$, which says "the farther along, the better". The shaped rewards are

$$\begin{aligned}
&\text{Start}: \quad \text{exit } 1 + 0 - 0 = 1, \qquad \text{go } 0 + 0.9 \cdot 1 - 0 = 0.9,\\
&\text{Middle}: \quad \text{exit } 2 + 0 - 1 = 1, \qquad \text{go } 0 + 0.9 \cdot 2 - 1 = 0.8,\\
&\text{Goal}: \quad \text{finish } 4 + 0 - 2 = 2.
\end{aligned}$$

Solving the shaped problem gives $Q'(\text{Start}, \cdot) = (1, 3.24)$, $Q'(\text{Middle}, \cdot) = (1, 2.6)$ and $Q'(\text{Goal}) = 2$ for (exit, go), which is Q^* minus Φ: the unshaped values are (1, 3.24), (2, 3.6) and 4. In every state the same action is best, so the optimal policy is still go, go, finish.

♦

Shaping preserves the optimal policy, but it changes how fast an agent finds it, and not always for the better. Figure 23.4 compares four versions of Q-learning on the grid with walls, twenty runs each. Let $d(s)$ be the number of steps from s to the goal if nothing were in the way (the Manhattan distance). The potential $\Phi(s) = \gamma^{d(s)-1}$ is the designer's estimate of the value of s; it is exact on this grid. With it, the median run is optimal after 1,000 steps instead of about 3,100. The potential $\Phi(s) = -d(s)/14$, a natural first attempt that rewards moving closer, makes learning *slower*: after 12,000 steps, 17 of the 20 runs are still not optimal. And the bonus near the goal, which is not a potential difference, teaches the agent never to finish.

Why would a potential that points toward the goal slow learning? Wiewiora (2003) showed that Q-learning with potential shaping, started from initial values Q_0, makes exactly the same choices as Q-learning without shaping started from $Q_0(s, a) + \Phi(s)$. Shaping *is* a choice of initial values. With $Q_0 = 0$, the potential $-d/14$ amounts to initial values below every true value, more so far from the goal. Pessimistic initial values discourage exploration (Chapter 2): an action that leaves the agent where it is, such as walking into a wall, is updated toward a value slightly higher than its pessimistic starting point, looks better than the untried actions, and gets chosen again. In the corner where the agent starts, two of the four actions bump into walls. The potential γ^{d-1} instead amounts to initial values that are exact for the best action and optimistic for the others. The practical rule: a good potential is an estimate of the value function, the closer the better, and erring on the high side is safer than erring on the low side.

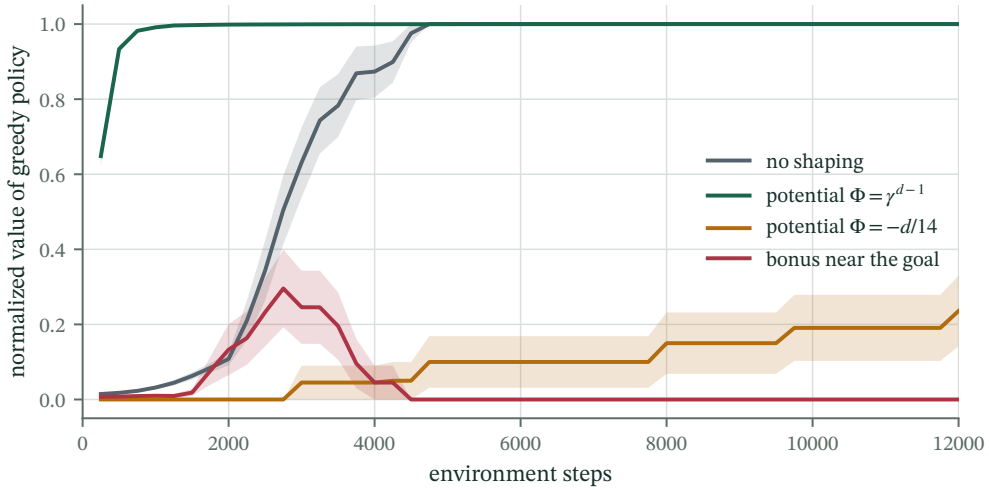


Figure 23.4. Q-learning on the 8×8 grid with walls with four rewards: no shaping (gray), potential shaping with $\Phi = \gamma^{d-1}$ (green) and with $\Phi = -d/14$ (amber), and a bonus of 0.1 for entering cells near the goal (red); d is the Manhattan distance to the goal. Lines are means of twenty runs, bands ± 1 standard error; the vertical axis is the exact value of the greedy policy under the true reward. Both potentials leave the optimal policy unchanged, yet one speeds learning and the other slows it; the bonus changes the optimal policy.

23.6.4 Why potential shaping works ●●

Theorem 23.1 Potential-based shaping preserves optimal policies

Let M be a finite discounted MDP with $\gamma \in [0, 1)$, and let $\Phi : \mathcal{S} \to \mathbb{R}$ satisfy $\Phi(s) = 0$ at every terminal state. Let M' be the same MDP with rewards $r'(s, a) = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\Phi(s') - \Phi(s)$. Then for every policy π and every state s,

$$V^{\pi}_{M'}(s) = V^{\pi}_{M}(s) - \Phi(s), \qquad Q^{\pi}_{M'}(s, a) = Q^{\pi}_{M}(s, a) - \Phi(s),$$

and the same holds for the optimal values. In particular M and M' have the same optimal policies.

Proof. Fix π and a start state $s_0 = s$. The expected shaped return over the first T steps is

$$\mathbb{E}_{\pi}\Big[\sum_{t=0}^{T-1} \gamma^t \big(r_t + \gamma\Phi(s_{t+1}) - \Phi(s_t)\big)\Big] = \mathbb{E}_{\pi}\Big[\sum_{t=0}^{T-1} \gamma^t r_t\Big] + \mathbb{E}_{\pi}\big[\gamma^T \Phi(s_T)\big] - \Phi(s_0),$$

because the potential terms telescope: $\sum_{t=0}^{T-1}\big(\gamma^{t+1}\Phi(s_{t+1}) - \gamma^t\Phi(s_t)\big) = \gamma^T\Phi(s_T) - \Phi(s_0)$. (Using the expected next potential $\sum_{s'} P(s' \mid s, a)\Phi(s')$ in r' instead of the realized $\Phi(s_{t+1})$ does not change the expectation, by the tower rule.) The state space is finite, so $|\Phi| \le \max_s |\Phi(s)| < \infty$ and the middle term tends to zero as $T \to \infty$; terminal states are absorbing with zero reward and, by assumption, zero potential, so they add nothing to either sum. Letting $T \to \infty$ gives $V^{\pi}_{M'}(s) = V^{\pi}_{M}(s) - \Phi(s)$. For Q, condition on the first action: $Q^{\pi}_{M'}(s, a) = r'(s, a) + \gamma \sum_{s'} P(s' \mid s, a)V^{\pi}_{M'}(s') = r(s, a) - \Phi(s) + \gamma \sum_{s'} P(s' \mid s, a)V^{\pi}_{M}(s') = Q^{\pi}_{M}(s, a) - \Phi(s)$.

Taking the maximum over policies on both sides of the identity for V (the shift $-\Phi(s)$ does not depend on π) gives $V^*_{M'} = V^*_M - \Phi$, and then $Q^*_{M'}(s,a) = Q^*_M(s,a) - \Phi(s)$ by the same conditioning. In each state, the actions that maximize $Q^*_{M'}(s,\cdot)$ are those that maximize $Q^*_M(s,\cdot)$, because the two differ by a constant in s. So the greedy policies, which are the optimal policies, coincide. ■

The proof shows where each condition is used. If a terminal state had a nonzero potential, the term $\gamma^T \Phi(s_T)$ would survive and depend on *when* the episode ends, which differs between policies; Exercise 23.9 shows that this can change the optimal policy of the Trail. In a finite-horizon problem the same argument works with time-dependent potentials Φ_t and $\Phi_H = 0$. Ng et al. (1999) also proved a converse: if the bonus F is not of the form (23.1), then there are transition probabilities and rewards for which shaping changes the optimal policy. Potential-based shaping is therefore exactly the set of bonuses that are safe for every problem. The same invariance explains why inverse reinforcement learning cannot recover a reward uniquely (Section 18.4): rewards that differ by a potential difference produce the same optimal behavior.

23.7 Safe and constrained reinforcement learning

23.7.1 What "safe" means

Many applications care about more than one quantity. A robot should finish its task quickly *and* avoid collisions; a data-center controller should save energy *and* keep temperatures within limits; a treatment policy should improve outcomes *and* limit side effects. One could add the costs to the reward with a penalty weight, but choosing the weight is hard: too small and the agent ignores the cost, too large and it never does anything useful. And the designer usually knows the limit ("at most one collision per hundred hours") more naturally than the exchange rate. *Safe reinforcement learning* covers several related goals (García and Fernández 2015; Amodei et al. 2016): respecting constraints in the final policy, respecting them *during* learning, avoiding rare catastrophic outcomes, and remaining safe when the world differs from the training environment. This section treats the first in detail and the others briefly.

23.7.2 Constrained MDPs

Definition 23.2 Constrained MDP

A *constrained MDP* (CMDP) adds to an MDP $(\mathcal{S}, \mathcal{A}, P, r, \gamma, d_0)$ a cost function $c(s,a) \geq 0$ and a budget d. With the discounted reward and cost

$$J_R(\pi) = \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big], \qquad J_C(\pi) = \mathbb{E}_\pi\Big[\sum_{t=0}^{\infty} \gamma^t c_t\Big], \qquad s_0 \sim d_0,$$

where $c_t = c(s_t, a_t)$, the goal is to find a policy that maximizes $J_R(\pi)$ subject to $J_C(\pi) \leq d$.

The theory of CMDPs is due largely to Altman (1999). Several constraints, each with its own cost and budget, are handled the same way. Note what the definition constrains: the *expected* discounted cost, not the cost of every episode. We return to this choice below.

Figure 23.5 shows the CMDP that we use throughout this section, the *hazard grid*. An agent starts at S and must reach the goal G; every move earns reward -1, and $\gamma = 0.95$. Six cells are hazards (think of a wet floor): entering one costs 1, but the cells are not terminal. The shortest path, straight along the bottom row, takes 6 steps and crosses three hazard cells, entered at times 1, 2 and 3; the shortest detour above the hazards takes 10 steps and enters none. Their values are

$$\text{direct:}\quad J_R = -\frac{1-\gamma^6}{1-\gamma} = -5.298, \quad J_C = \gamma + \gamma^2 + \gamma^3 = 2.710,$$
$$\text{detour:}\quad J_R = -\frac{1-\gamma^{10}}{1-\gamma} = -8.025, \quad J_C = 0.$$

With a budget $d = 1$, the direct path is too costly and the detour is wasteful. Every other deterministic policy either exceeds the budget as well (any path shorter than 10 steps enters at least three hazards) or takes at least as long as the detour.

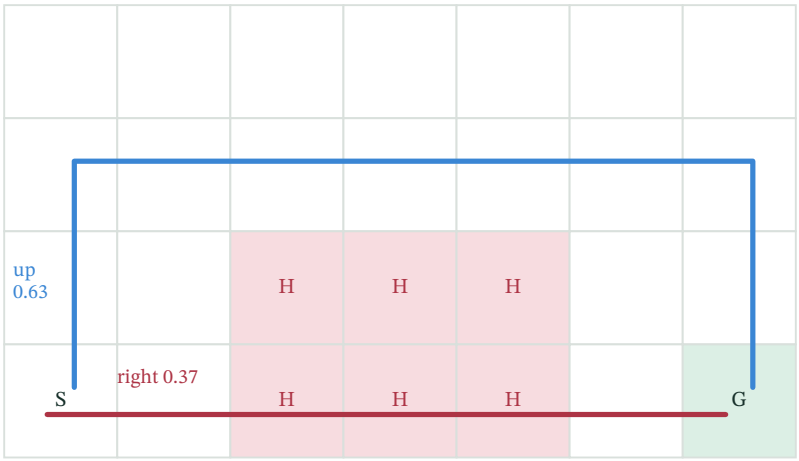


Figure 23.5. The hazard grid. The agent starts at S and must reach G; each move earns -1, and entering a hazard cell (H) costs 1. The direct path (red) is fast but crosses three hazards; the detour (blue) avoids them but takes four extra steps. With a budget of 1 on the expected discounted cost, the optimal policy randomizes at the start: up with probability 0.63, right with probability 0.37.

The best policy mixes the two. If the agent takes the direct path with probability α and the detour otherwise, its cost is $2.710\,\alpha$ and its return $-5.298\,\alpha - 8.025\,(1-\alpha)$. Using the whole budget, $\alpha = 1/2.710 = 0.369$, gives $J_R = -7.019$, and this is the optimum of the CMDP (we verify it below, and the lab confirms it with a linear program). The optimal constrained policy is *stochastic*: it flips a biased coin in the start state and acts deterministically afterward.

This is a general feature. In an unconstrained MDP some deterministic policy is always optimal (Chapter 3). With constraints that is no longer true, and the reason is geometric. The expected discounted visit counts of a policy, its *occupancy measure* $\nu_\pi(s,a) = \sum_t \gamma^t \Pr_\pi(s_t = s, a_t = a)$, form a convex polytope, and both $J_R = \sum_{s,a} \nu_\pi(s,a)\, r(s,a)$ and $J_C = \sum_{s,a} \nu_\pi(s,a)\, c(s,a)$ are linear in it (Section 3.10.3). A CMDP is therefore a linear program over this polytope. Without the constraint, a linear function is maximized at a vertex, and the vertices are the deterministic policies. The constraint cuts the polytope with a half-space, and the optimum can lie on the cut, between two vertices. In fact some optimal policy randomizes in at most as many states as there

are constraints (Altman 1999).

PITFALL

The constraint $J_C(\pi) \le d$ limits the *average* cost. The optimal policy of the hazard grid takes the dangerous path in 37% of its episodes, every time crossing three hazards. If the cost stands for something that must never happen, an expected-cost constraint is the wrong specification; risk-sensitive objectives and constraints on the probability of failure address this, at a computational price.

23.7.3 The Lagrangian method

The standard way to solve a CMDP without writing down the linear program, and the only practical way when the policy is a neural network, is to move the constraint into the objective with a *Lagrange multiplier* $\lambda \ge 0$ (Section A.5):

$$L(\pi, \lambda) = J_R(\pi) - \lambda \big(J_C(\pi) - d\big). \tag{23.2}$$

For a fixed λ, maximizing L over π is an ordinary MDP with reward $r - \lambda c$: a cost of one unit is worth λ units of reward. The multiplier is the exchange rate that the designer did not want to choose by hand, and the method chooses it automatically: raise λ while the constraint is violated, lower it when there is slack.

Algorithm 23.2 Lagrangian method for a constrained MDP

Input: budget d; policy step size η; multiplier step size η_λ; initial policy parameters θ_0; $\lambda_0 = 0$.

1. For $k = 0, 1, 2, \ldots$:
   1. Collect experience with π_{θ_k}; estimate $J_C(\pi_{\theta_k})$ and the advantages A_R and A_C for reward and cost.
   2. Policy step: improve the policy for the reward $r - \lambda_k c$, for example $\theta_{k+1} \leftarrow \theta_k + \eta\, \widehat{\nabla}_\theta \big[J_R(\theta_k) - \lambda_k J_C(\theta_k)\big]$, or a PPO update with advantages $A_R - \lambda_k A_C$.
   3. Multiplier step: $\lambda_{k+1} \leftarrow \max\{0,\ \lambda_k + \eta_\lambda\, (\hat{J}_C - d)\}$.

Output: the last policy, or the average of the policies (see the text).

The policy step is any of the policy-improvement methods of Part IV applied to the penalized reward; the multiplier step is gradient descent on L in λ, since $\partial L / \partial \lambda = -(J_C - d)$, projected onto $\lambda \ge 0$. The multiplier usually moves on a slower timescale than the policy, which is the setting of two-timescale stochastic approximation (Appendix B). With PPO as the policy step this is the "PPO-Lagrangian" baseline of the Safety Gym benchmark (Ray et al. 2019); Tessler et al. (2019) analyze a closely related method, reward-constrained policy optimization, with a convergence argument based on multi-timescale stochastic approximation.

Figure 23.6 runs Algorithm 23.2 on the hazard grid with exact policy evaluation, so that what we see is the method itself and not sampling noise. The policy is a softmax over the four actions

in each state, and the policy step is the natural policy gradient step of Section 12.4, which for a tabular softmax policy is $\theta(s,a) \leftarrow \theta(s,a) + \frac{\eta}{1-\gamma}\big(A_R(s,a) - \lambda_k A_C(s,a)\big)$, with $\eta = 0.02$ and $\eta_\lambda = 0.1$. With the plain multiplier step (gray), the method never settles. While λ is below the exchange rate at which the two paths are equally good, the policy drifts to the direct path and the cost rises to 2.7; the violation pushes λ up, the policy swings to the detour, the cost drops to 0, and λ falls again. The cycle repeats about every 48 iterations. Its running average (amber) does approach the constrained optimum, slowly: after 2,000 iterations the average cost is 0.87 and the average return −7.16.

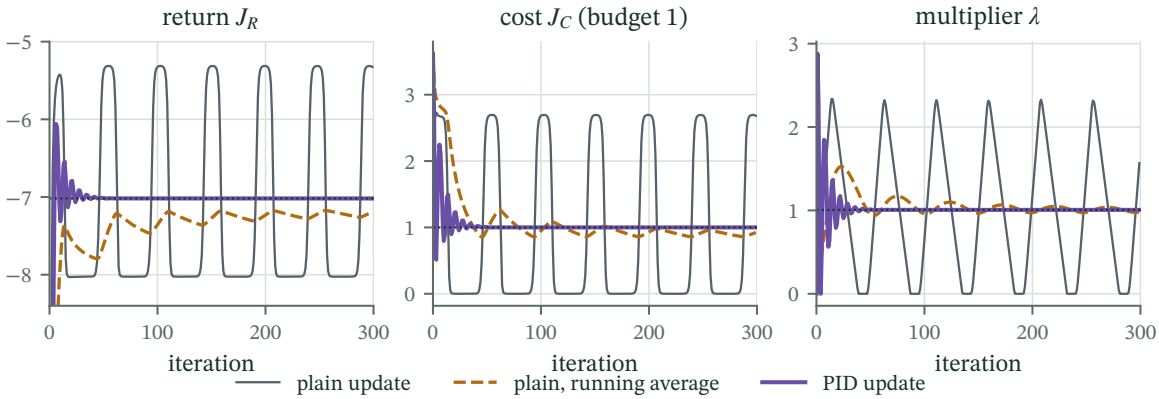


Figure 23.6. The Lagrangian method on the hazard grid with budget 1, using exact natural policy gradient steps. Gray: the plain multiplier update, under which the policy keeps jumping between the direct path and the detour. Amber, dashed: running averages of the plain method. Violet: a multiplier set by a PID controller, which settles within 1% of the budget after 44 iterations. Dotted lines: the exact optimum $J_R = -7.019$, the budget, and $\lambda^* = 1.006$. The plot shows the first 300 of 2,000 iterations.

Oscillation is the typical failure of Lagrangian methods: the multiplier responds to the *accumulated* violation, like the integral term of a controller, and so it overshoots. Stooke et al. (2020) made the analogy precise and proposed *PID Lagrangian* methods, which add a term proportional to the current violation $J_C - d$ and one proportional to its rate of increase. The violet curves use the multiplier $\lambda_{k+1} = \max\{0,\, K_P\, e_k + \eta_\lambda I_k\}$ with $e_k = J_C(\pi_k) - d$, the accumulated violation $I_k = \max\{0, I_{k-1} + e_k\}$ and $K_P = 1$. The method stays within 1% of the budget after 44 iterations and converges to the exact optimum: $J_R = -7.019$, $J_C = 1.000$ and $\lambda = 1.006$. Two other remedies are common. One can report the average of the iterates, for which convergence guarantees exist (Ding et al. 2020). Or one can add a small entropy bonus, which makes the best response a smooth function of λ at the price of a small bias (try it in the lab).

The converged multiplier has a meaning of its own. At $\lambda^* = 1.006$ the two paths are equally good for the reward $r - \lambda^* c$: $-5.298 - 1.006 \times 2.710 = -8.025$, exactly the return of the detour. So λ^* is the *price* of the constraint: relaxing the budget by a small amount δ raises the best achievable return by about $\lambda^* \delta$.

23.7.4 Lagrangian duality for CMDPs ●●

Why does the Lagrangian method find the *constrained* optimum, rather than something else? Define the *dual function*

$$g(\lambda) = \max_{\pi} L(\pi, \lambda) = \max_{\pi} \big[J_R(\pi) - \lambda\,(J_C(\pi) - d)\big], \qquad \lambda \geq 0.$$

For each λ it is computed by solving one unconstrained MDP. For every feasible π (one with $J_C(\pi) \leq d$) and every $\lambda \geq 0$, $J_R(\pi) \leq L(\pi, \lambda) \leq g(\lambda)$; so the best feasible return is at most $\min_{\lambda \geq 0} g(\lambda)$. This is *weak duality*. For CMDPs the inequality is an equality.

Theorem 23.3 No duality gap for constrained MDPs

Let a finite CMDP with $\gamma \in [0, 1)$ and one constraint have a feasible policy. Then g is convex and piecewise linear on $[0, \infty)$, it attains its minimum at some $\lambda^* \geq 0$, and

$$\max_{\pi : J_C(\pi) \leq d} J_R(\pi) = \min_{\lambda \geq 0} g(\lambda) = g(\lambda^*).$$

Moreover, if $\lambda^* > 0$, there are policies π_1 and π_2, both optimal for the reward $r - \lambda^* c$, with $J_C(\pi_1) \geq d \geq J_C(\pi_2)$, and the stationary policy whose occupancy measure is $\alpha\, \nu_{\pi_1} + (1 - \alpha)\, \nu_{\pi_2}$, with $\alpha \in [0, 1]$ chosen so that its cost equals d, is an optimal policy of the CMDP. If $\lambda^* = 0$, some optimal policy of the unconstrained MDP satisfies the constraint and is optimal for the CMDP.

Proof. Write ν_π for the occupancy measure of π and $\mathcal{K}$ for the set of occupancy measures of all policies. Two facts from the linear-programming view of MDPs (Section 3.10.3; see also Puterman (1994), sec. 6.9, and Altman (1999)) are used: $\mathcal{K}$ is a convex polytope whose vertices are occupancy measures of deterministic stationary policies, and every $\nu \in \mathcal{K}$ is the occupancy measure of the stationary policy $\pi_\nu(a \mid s) = \nu(s, a) / \sum_{a'} \nu(s, a')$ (any action distribution in states with zero occupancy). Then $J_R(\pi) = \langle \nu_\pi, r \rangle$ and $J_C(\pi) = \langle \nu_\pi, c \rangle$, and $L(\pi, \lambda) = \langle \nu_\pi, r - \lambda c \rangle + \lambda d$.

Shape of g. A linear function on a polytope is maximized at a vertex, so $g(\lambda) = \max_v \big[\langle v, r \rangle - \lambda(\langle v, c \rangle - d)\big]$, a maximum over the finitely many vertices v of $\mathcal{K}$ of functions that are affine in λ. Hence g is convex and piecewise linear. The slope of the piece of vertex v is $d - \langle v, c \rangle$. For large λ the active pieces are those of vertices with the smallest cost, $c_{\min} = \min_{\nu \in \mathcal{K}} \langle \nu, c \rangle$, and their slope $d - c_{\min}$ is nonnegative because a feasible policy exists. A convex piecewise-linear function whose last slope is nonnegative attains its minimum on $[0, \infty)$, at some λ^*.

No gap. Let $\mathcal{A}^* \subseteq \mathcal{K}$ be the set of maximizers of $\langle \nu, r - \lambda^* c \rangle$ over $\mathcal{K}$; it is a face of the polytope, so it is convex. Each $\nu \in \mathcal{A}^*$ gives an affine function of λ that touches g at λ^*, so the one-sided derivatives of g at λ^* are $g'_-(\lambda^*) = d - \max_{\nu \in \mathcal{A}^*} \langle \nu, c \rangle$ and $g'_+(\lambda^*) = d - \min_{\nu \in \mathcal{A}^*} \langle \nu, c \rangle$ (for $\lambda^* = 0$ only the right derivative is defined). Since λ^* minimizes g: if $\lambda^* > 0$, then $g'_-(\lambda^*) \leq 0 \leq g'_+(\lambda^*)$, so $\mathcal{A}^*$ contains ν_1 with $\langle \nu_1, c \rangle \geq d$ and ν_2 with $\langle \nu_2, c \rangle \leq d$, and by convexity it contains the mixture $\bar{\nu} = \alpha \nu_1 + (1 - \alpha) \nu_2$ with $\langle \bar{\nu}, c \rangle = d$; if $\lambda^* = 0$, then $g'_+(0) \geq 0$ and $\mathcal{A}^*$ contains $\bar{\nu}$ with $\langle \bar{\nu}, c \rangle \leq d$. In both cases $\lambda^*(\langle \bar{\nu}, c \rangle - d) = 0$ (*complementary slackness*), so the policy $\bar{\pi} = \pi_{\bar{\nu}}$ is feasible and

$$J_R(\bar{\pi}) = \langle \bar{\nu}, r \rangle - \lambda^*(\langle \bar{\nu}, c \rangle - d) = g(\lambda^*).$$

By weak duality no feasible policy has a return above $g(\lambda^*)$, so $\bar{\pi}$ is optimal and the two sides of the displayed equation are equal. Taking $\pi_1 = \pi_{\nu_1}$ and $\pi_2 = \pi_{\nu_2}$ (and, if $\lambda^* = 0$, $\bar{\pi}$ itself) gives the last statement. ■

The proof is also an algorithm, which `rlcourse.safe.solve_cmdp` implements: find λ^* by bisection on the sign of $d - J_C$ of the best response (the slope of g), take deterministic best responses just below and just above λ^*, and mix their occupancy measures. On the hazard grid it returns $\lambda^* = 1.006$ and the coin-flipping policy of Figure 23.5, and a linear-programming solver gives the same answer. The theorem also explains why the plain Lagrangian method oscillates: the best response to λ jumps from the direct path to the detour as λ crosses λ^*, and no deterministic best response satisfies the constraint with equality. For policies parameterized by neural networks, J_R and J_C are not linear in the parameters, but Paternain et al. (2019) proved that the duality gap is still zero for the problem over all policies, and that parameterizations rich enough to approximate every policy have a small gap. Ding et al. (2020) analyzed the natural policy gradient primal-dual method and bounded the optimality gap and the constraint violation, averaged over its iterates.

23.7.5 Constrained policy optimization and safe exploration

The Lagrangian method satisfies the constraint only at convergence, and only on average over its iterates. *Constrained policy optimization* (CPO) (Achiam et al. 2017) aims to keep every iterate feasible. It extends the trust-region step of TRPO (Chapter 12): it maximizes a surrogate of the reward subject to the KL trust region and to a linear approximation of the cost constraint, using a bound in the spirit of the performance difference lemma to control how much the true cost can differ from its approximation. CPO's guarantee is approximate, because the constraint is linearized and estimated from samples, and in the Safety Gym benchmark Lagrangian versions of PPO and TRPO were competitive with it (Ray et al. 2019).

Constraints on the final policy do not protect anything while the agent learns, and an exploring agent will try dangerous actions. *Safe exploration* methods restrict the agent's actions during learning using prior knowledge. A *safety layer* corrects each proposed action as little as possible so that a learned or known model predicts that the constraints remain satisfied (Dalal et al. 2018). A *shield* (Alshiekh et al. 2018) uses a formal model of the safety-relevant dynamics and a specification in temporal logic to block actions that could lead to violations, at the price of needing that model. In practice, the most reliable safe exploration is to explore somewhere else: in a simulator (Section 23.8), or from logged data (Chapter 17).

23.7.6 Risk-sensitive objectives ●●

The expected return treats a policy that earns 4 for sure the same as one that earns 10 with probability 0.9 and -50 with probability 0.1; both have mean 4. Risk-sensitive reinforcement learning optimizes a criterion that looks at the whole distribution of the return. The most widely used is the *conditional value at risk*. For a return G and a level $\alpha \in (0, 1]$, the *value at risk* $\mathrm{VaR}_\alpha(G)$ is the α-quantile of G, and $\mathrm{CVaR}_\alpha(G)$ is the mean of the worst α-fraction of outcomes; for a continuous distribution, $\mathrm{CVaR}_\alpha(G) = \mathbb{E}[G \mid G \le \mathrm{VaR}_\alpha(G)]$. A useful formula, due to

Rockafellar and Uryasev (2000), holds for every distribution:

$$\mathrm{CVaR}_\alpha(G) = \max_{\nu \in \mathbb{R}} \Big\{ \nu - \frac{1}{\alpha}\, \mathbb{E}\big[(\nu - G)^+\big] \Big\}, \tag{23.3}$$

where $x^+ = \max\{x, 0\}$, and the maximum is attained at $\nu = \mathrm{VaR}_\alpha(G)$ (Exercise 23.12). It turns CVaR into an expectation plus a one-dimensional optimization, which sampling-based methods exploit (Tamar et al. 2015; Chow et al. 2018).

For the risky policy above, $\mathrm{CVaR}_{0.1} = -50$ (the worst 10% is the disaster), $\mathrm{CVaR}_{0.2} = \frac{1}{0.2}(0.1 \cdot (-50) + 0.1 \cdot 10) = -20$, and $\mathrm{CVaR}_{0.5} = -2$; at $\alpha = 1$ CVaR is the mean, 4. A CVaR-maximizing agent prefers the sure 4 at every level $\alpha < 1$. For the Trail's uniform policy, whose return is 1, 1.8 or 3.24 with probabilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{4}$, the worst half of the outcomes is the exit at Start, so $\mathrm{CVaR}_\alpha = 1$ for $\alpha \le 0.5$, and $\mathrm{CVaR}_{0.75} = (0.5 \cdot 1 + 0.25 \cdot 1.8)/0.75 = 1.267$.

Optimizing CVaR is harder than optimizing a mean. CVaR is not an expectation of a sum of rewards, so it does not satisfy a Bellman equation in the original state, and the best policy can depend on the reward collected so far. Methods such as those of Chow et al. (2018) optimize over the policy and the threshold ν of (23.3) together, which amounts to adding the accumulated reward to the state. Distributional reinforcement learning (Section 10.8) learns the whole return distribution and makes such criteria easy to evaluate, although acting greedily with respect to a risk measure of a learned distribution is not in general optimal for that measure.

23.8 From simulation to reality

Most successful applications of reinforcement learning to physical systems train in simulation. Simulators are fast, parallel, safe and resettable; the real world is none of these. The difficulty is the *reality gap*: a policy that exploits details of the simulator (an unrealistically grippy floor, a perfectly known mass, noise-free sensors) can fail when those details change. A capable optimizer finds such details just as it finds loopholes in a reward. Three families of techniques close the gap.

System identification makes the simulator match reality: measure or fit its parameters (masses, friction, motor constants, delays) to data from the real system. Some parts of a robot are hard to model from first principles; Hwangbo et al. (2019) replaced the simulated model of a legged robot's actuators by a neural network fitted to measurements of the real actuators, and trained locomotion policies in the resulting simulator that worked on the real robot.

Domain randomization gives up on matching reality exactly and trains on a whole distribution of simulators instead. If ξ denotes the simulator's parameters and $J_\xi(\pi)$ the return in the simulator with parameters ξ, the agent maximizes

$$\mathbb{E}_{\xi \sim p}\big[J_\xi(\pi)\big],$$

in the hope that the real world looks like one more sample from p. Tobin et al. (2017) randomized the textures, lighting and camera positions of a simulator and trained object detectors, used for robotic grasping, that worked on real images without ever seeing one. Peng et al. (2018) randomized the dynamics (masses, friction, damping, latency) and transferred a pushing policy

for a robot arm to the real arm; their policy was recurrent, so it could infer the parameters of the current episode from its own history. OpenAI's robot hand that solved a Rubik's cube was trained with a schedule that gradually widened the randomization as the policy improved (OpenAI et al. 2019). The quadrotor that beat human champions in drone racing combined simulation with models of the residual errors fitted to real flight data (Kaufmann et al. 2023), and policies for legged robots are now routinely trained in minutes in massively parallel simulation (Rudin et al. 2022).

Robust and adaptive design goes further than randomization: it can optimize the worst case over plausible parameters instead of the average, or train policies that adapt online. The hazard grid shows the stakes for constraints in particular. The optimal constrained policy for the deterministic grid has expected cost exactly 1. Deployed in a world where each move slips sideways with probability 0.1, the same policy has expected cost 1.13; with slip probability 0.2, 1.27. The constraint is violated by 13% and 27%. The optimal constrained policy for the slip-0.2 world, in contrast, meets the budget in all three worlds (costs 0.62, 0.81 and 1.00), at the price of a lower return in the deterministic world (-7.41 instead of -7.02). Robustness costs performance in the nominal case, and the designer must decide how much uncertainty to plan for.

Simulation is not the only route. In many applications, such as healthcare, recommendation and industrial control, there is no simulator but there are logs of past decisions. Learning from such data without further interaction is *offline reinforcement learning* (Chapter 17), and evaluating a new policy from old data is *off-policy evaluation* (Section 17.3). Dulac-Arnold et al. (2021) list the further challenges that real systems bring, including delays, partial observability, constraints, multiple objectives and the need to explain decisions, and propose benchmark variants that include them.

23.9 Compute and scale ●●

The cost of a deep RL experiment is measured in environment steps, gradient updates and wall-clock time, and the three are traded against each other. The first number to know is the *throughput*: environment steps per second, together with where the time goes (stepping the simulator, running the policy network, updating it). A simple profile often reveals that a fast learner waits for a slow, single-threaded simulator.

The classic remedy is parallelism. Synchronous methods such as A2C and PPO step many environment copies in lockstep (Section 23.2.3); libraries such as EnvPool (Weng et al. 2022) run thousands of copies in compiled code. *Distributed actor–learner* architectures separate the two jobs: many *actors* generate experience with a recent copy of the policy and send it to one or a few *learners* that update the network on accelerators. Because the actors' policies lag behind the learner's, the data are slightly off-policy; IMPALA corrects for the lag with the V-trace importance weights (Espeholt et al. 2018) of Section 7.8. Off-policy value-based agents distribute experience collection in the same way and share a large replay buffer (Chapter 10).

A more recent remedy removes the boundary between simulator and learner. Physics simulators written for accelerators, such as Brax (Freeman et al. 2021) and Isaac Gym (Makoviychuk et al. 2021), run thousands of environments in parallel on one GPU, so that experience never

leaves the device; whole training loops, environment included, can then be compiled together. Throughputs of millions of steps per second on a single machine have turned experiments that once needed a cluster into experiments that run in minutes.

Does more compute buy better agents? More environment steps usually help, if the algorithm can use them. Bigger networks are less straightforward than in supervised learning: in many deep RL settings, simply widening or deepening the network does not help and can hurt, because of instability and loss of plasticity (Section 24.5). Several lines of recent work have found recipes under which scale does pay, combining larger networks with normalization, periodic resets of some parameters, or careful choice of the number of updates per environment step (Chapter 10); world-model agents such as DreamerV3 reported that larger models improved both final performance and data efficiency (Hafner et al. 2025). Small-scale experiments remain valuable: many scientific questions about RL can be answered on small environments where dozens of runs are cheap (Obando-Ceron and Castro 2021), which also makes the statistics of Section 23.4 affordable.

23.10 Reproducibility and ethics

A result is reproducible when someone else, with the same code and data, obtains the same numbers, and *replicable* when an independent implementation reaches the same conclusions. Reinforcement learning makes both hard: results depend on seeds, library versions, hardware and many unreported details. Exact reproducibility is often impossible even with fixed seeds, because parallel computation on GPUs is not deterministic; what can be reproduced is the *distribution* of results, which is one more reason to report many runs. The reporting checklist of Section 23.4.7 is the core of good practice. Releasing code, configurations and the scores of every run costs little and lets others check, reuse and extend the work; reproducibility programs at machine-learning conferences ask for exactly this (Pineau et al. 2021). Libraries with careful, tested implementations, such as Stable-Baselines3 (Raffin et al. 2021) and the single-file implementations of CleanRL (Huang, Dossa, Ye, et al. 2022), make baselines more trustworthy.

Experiments also have costs outside the lab. Training large models consumes energy and produces emissions; Strubell et al. (2019) estimated these costs for natural-language models, and Henderson et al. (2020) proposed tools for measuring and reporting the energy and carbon footprint of machine-learning experiments, including reinforcement learning. Reporting compute is a step toward taking these costs into account, and so is choosing the smallest experiment that answers the question.

Finally, reinforcement learning optimizes whatever it is given, which makes the choice of objective an ethical choice. A recommender system that maximizes engagement optimizes engagement, whether or not engagement serves its users. Systems that learn from and act on people raise questions of consent, fairness and accountability, and the same methods that control a robot or a power grid could be applied to harmful ends; research on autonomous decision making has dual uses. Clear specifications, constraints that encode limits, human oversight, staged deployment and monitoring after deployment are part of doing reinforcement learning well, not afterthoughts (Amodei et al. 2016; Russell and Norvig 2021).

23.11 Summary

- Reinforcement learning is hard to get right because the data depend on the learner, results are sensitive to small details, and bugs are silent: an agent with a bug still learns something.
- Implement the environment loop with care: bootstrap at truncations but not at terminations, know the reset convention of your vector environments, save and freeze normalization statistics, and log the diagnostics that explain a learning curve (entropy, KL, explained variance, gradient norms, value magnitudes).
- Debug from small to large, checking each component against known answers; on the Trail and on grid worlds every quantity can be computed exactly.
- Runs of one algorithm vary a lot. The unit of evaluation is the run; report intervals (the t interval for means of few runs, the bootstrap for other statistics and for aggregates), robust aggregates such as the interquartile mean, and performance profiles.
- Few runs detect only large effects: 17 runs per algorithm are needed to detect a difference of one standard deviation with 80% power, and 64 for half a standard deviation. Compare algorithms with equal tuning budgets and without selecting on the test data.
- Hyperparameters matter, and the best values differ between tasks; random search beats grid search, and sensitivity curves are more informative than a single tuned number.
- Rewards are specifications that agents take literally. Potential-based shaping never changes the optimal policy, and it acts like a choice of initial values; other bonuses can be farmed.
- Constrained MDPs limit expected costs. Their optimal policies may randomize; the Lagrangian method finds them by adjusting a multiplier that has the meaning of a price, and the dual problem has no gap (●●). Plain multiplier updates oscillate; PID updates, averaging or entropy regularization fix this.
- Policies move from simulation to reality through system identification, domain randomization and robust design; constraints satisfied in a nominal simulator can be violated in a slightly different world.

The next chapter looks beyond the current state of practice, to the open problems of the field and to how research on them is done.

23.12 Further Reading

Engineering. Henderson et al. (2018) is the paper that made the community take reproducibility seriously; read it with Engstrom et al. (2020) and Andrychowicz et al. (2021), which measure which implementation details matter for policy-gradient methods, and with the catalogue of PPO details by Huang, Dossa, Raffin, et al. (2022). Pardo et al. (2018) is the reference on time limits. The documentation of Gymnasium (Towers et al. 2024), Stable-Baselines3 (Raffin et al. 2021) and CleanRL (Huang, Dossa, Ye, et al. 2022) is worth reading for its treatment of vector environments, wrappers and normalization.

Evaluation. R. Agarwal et al. (2021) is required reading before reporting deep RL results, and its companion library implements the IQM, stratified bootstrap and performance profiles of Section 23.4. Colas et al. (2018) explain power analysis for RL experiments, Patterson et al. (2024)

give a thorough treatment of experimental design in reinforcement learning, including sensitivity analysis and the choice of baselines, and Jordan et al. (2020) and Chan et al. (2020) propose measures of performance and reliability across environments. Efron and Tibshirani (1993) is the classic introduction to the bootstrap.

Reward design. Ng et al. (1999) introduced potential-based shaping and Wiewiora (2003) showed its equivalence to initialization. Krakovna et al. (2020) collect examples of specification gaming, and Amodei et al. (2016) frame reward hacking as one of several concrete problems of AI safety.

Safe RL. Altman (1999) is the standard monograph on constrained MDPs, and García and Fernández (2015) a survey of safe reinforcement learning. For deep RL, start with Achiam et al. (2017) (CPO), Ray et al. (2019) (Safety Gym and its Lagrangian baselines) and Stooke et al. (2020) (PID Lagrangian methods). Paternain et al. (2019) and Ding et al. (2020) are good entry points into the theory of primal-dual methods (●●●). Chow et al. (2018) treat risk-constrained RL with CVaR.

Simulation to reality. Tobin et al. (2017) and Peng et al. (2018) introduced visual and dynamics randomization; OpenAI et al. (2019) show how far randomization can be pushed; Dulac-Arnold et al. (2021) survey the challenges of real-world RL.

23.13 Exercises

Exercise 23.1 ● Diagnosing a truncation bug

An agent's critic is trained on a task with reward +1 at every step, no terminal states, a time limit of $H = 10$ steps and $\gamma = 0.9$. Its targets are computed as `r + gamma * (1 - done) * V(s2)` with `done = terminated or truncated`. (a) What value does TD(0) converge to, and what is the true value? (b) Fix the line of code. (c) A colleague argues that the time limit is "part of the task" and the code is therefore correct. What would have to be added to the state for the colleague's view to be consistent?

Exercise 23.2 ● Reading learning curves

For each description, give two likely causes and the diagnostic you would look at first. (a) PPO's return rises for 200,000 steps, then falls to the random level within 20,000 steps, while the approximate KL between updates spikes. (b) DQN's return never rises above that of a random policy, and its Q-values grow steadily to 10^4 in a task with rewards in $[0, 1]$ and $\gamma = 0.99$. (c) An agent's training return is high, but its evaluation return is much lower. (d) The mean over five runs rises smoothly and slowly, but every single run jumps abruptly from failure to success.

Exercise 23.3 ● The bootstrap by hand

Three runs scored 1, 2 and 6. (a) List the distinct values of the mean of a bootstrap resample (three scores drawn with replacement) and their probabilities. (b) Compute the mean and the standard

deviation of this bootstrap distribution, and compare them with the sample mean and with $s/\sqrt{3}$, where s is the sample standard deviation. (c) Give the 80% percentile bootstrap interval, from the 10% to the 90% quantile of the bootstrap distribution. Why should you not trust it?

Exercise 23.4 ● Mean, median and IQM

Eight runs (two tasks, four runs each, already normalized) scored 0.10, 0.20, 0.25, 0.30, 0.40, 0.50, 0.90 and 5.00. (a) Compute the mean, the median and the interquartile mean. (b) Replace 5.00 by 0.95 and recompute. (c) Which statistic would you report, and why might a reader still want to see the per-task scores?

Exercise 23.5 ● How many seeds?

In pilot runs, an algorithm's final return has a standard deviation of about 40 across seeds. You want to detect an improvement of 20 in the mean final return with a two-sided test at the 5% level and 80% power. (a) What is the effect size? (b) How many runs per algorithm do you need (use Table 23.3 and the normal approximation)? (c) With ten runs per algorithm, what is the smallest effect size d that you could detect with 80% power?

Exercise 23.6 ●● Welch's test by hand

Algorithm A scored 12, 15, 9, 14 and algorithm B scored 8, 10, 7 on four and three runs. Compute Welch's statistic t and its degrees of freedom ν, and use the 97.5% quantile of the t distribution with ν degrees of freedom (about 2.60 here) to decide whether the difference is significant at the 5% level. Why are the degrees of freedom not $4 + 3 - 2 = 5$?

Exercise 23.7 ●● The best checkpoint is too good

Each of k checkpoints of a run is equally good, and its evaluation score is the true score plus independent noise σZ_i with Z_i standard normal. (a) Show that reporting the best evaluation overstates the true score by $\sigma\,\mathbb{E}[\max_i Z_i]$ on average, and compute $\mathbb{E}[\max(Z_1, Z_2)] = 1/\sqrt{\pi}$. (b) For $k = 10$ the bias is 1.54σ. If evaluation uses ten episodes with returns of standard deviation 50, how large is the bias in return units? (c) Describe two protocols that remove the bias.

Exercise 23.8 ● Bonuses that change the answer

Take the Trail with $\gamma = 0.4$, for which the optimal policy exits at Start. (a) A designer adds a bonus $b > 0$ to the reward of every go action. Find the smallest b that changes the optimal policy, and say which policy becomes optimal. (b) Replace the bonus by the potential-based term (23.1) with $\Phi(\text{Start}) = 0$, $\Phi(\text{Middle}) = b$, $\Phi(\text{Goal}) = 2b$ and $\Phi(\text{End}) = 0$. Compute the shaped rewards and check that the optimal policy does not change for any b.

Exercise 23.9 ●● Why terminal potentials must vanish

(a) On the Trail with $\gamma = 0.9$, use the "potential" $\Phi = 0$ in Start, Middle and Goal and $\Phi(\text{End}) = c$. Compute the shaped return of each of the three paths, and find the values of c for which the optimal policy of the shaped problem exits at Start. (b) Explain which step of the proof of Theorem 23.1 fails. (c) State and prove the version of Theorem 23.1 for a finite-horizon problem with time-dependent potentials Φ_t, $t = 0, \dots, H$, and $\Phi_H = 0$.

Exercise 23.10 ●● The dual of a two-state CMDP

A CMDP has states A (the start) and B, and $\gamma = 0.9$. In A, the action risky earns reward 2, costs 1 and ends the episode; the action safe earns 0, costs 0 and moves to B. In B the only action earns 1 and ends the episode. The budget is $d = 0.5$. (a) Compute J_R and J_C of the two deterministic policies. (b) Compute the dual function $g(\lambda)$, find its minimizer λ^* and the value $g(\lambda^*)$. (c) Find the optimal policy and check complementary slackness. (d) Show that no deterministic policy is optimal.

Exercise 23.11 ● CVaR of a distribution

A risky policy earns 10 with probability 0.9 and -50 with probability 0.1; a safe policy earns 3 for sure. (a) Compute CVaR_α of the risky return for $\alpha = 0.1, 0.2, 0.5, 1$. (b) For which levels α does a CVaR-maximizing agent prefer the safe policy? (c) Compute $\text{CVaR}_{0.75}$ of the return of the Trail's uniform policy with $\gamma = 0.9$.

Exercise 23.12 ●● The Rockafellar–Uryasev formula

Let G take finitely many values. (a) Show that the function $h(\nu) = \nu - \frac{1}{\alpha}\mathbb{E}[(\nu - G)^+]$ is concave and piecewise linear, and that its slope is $1 - \Pr(G < \nu)/\alpha$ between support points. (b) Deduce that h is maximized at $\nu = \text{VaR}_\alpha(G)$. (c) Show that $h(\text{VaR}_\alpha(G))$ equals the mean of the worst α-fraction of outcomes, splitting an atom at the boundary if necessary, which proves (23.3).

Exercise 23.13 ● Evaluate two agents properly

Extend Task 4 of Lab 23.1 to thirty runs per agent and task (`safe.compare_agents(n_seeds=30)`). Report the IQM of each agent with a 95% stratified-bootstrap interval, the probability of improvement with an interval, and Welch's test on each task with a Bonferroni correction for five tests. Do your conclusions differ from those with ten runs? Write the results paragraph you would put in a paper, following the checklist of Section 23.4.7.

Exercise 23.14 ●● The reality gap for a constraint

Using `safe.hazard_grid(slip=...)` and `safe.solve_cmdp`, (a) evaluate the optimal constrained policy of the deterministic hazard grid in grids with slip probabilities 0.1 and 0.2; (b) find the optimal constrained policy for slip 0.2 and evaluate it in all three grids; (c) explain the trade-off you observe, and propose a way to choose a policy when the slip probability is only known to lie in $[0, 0.2]$.

Exercise 23.15 ●●● Benchmarks that measure what matters

Choose a benchmark suite that is widely used in deep RL (for example the Atari games, the MuJoCo locomotion tasks, or the Procgen games). Discuss what it rewards and what it ignores: sample efficiency versus final performance, generalization to new levels or dynamics, robustness to the seed, compute cost, safety. Propose a protocol or a set of measures that would make a result on the suite more informative, and discuss what your proposal would cost to run. Support your argument with the literature.

Exercise 23.16 ●●● Safety during training

The Lagrangian method of Algorithm 23.2 satisfies the constraint only at convergence, and Figure 23.6 shows that its iterates can violate it badly along the way. Discuss what guarantees one could want during training (for example a bound on the total cost incurred while learning), why they are hard to obtain without prior knowledge, and what approaches the literature offers. Which of them would you trust on a real robot?

CHAPTER 24

Frontiers and Open Problems

WHAT YOU WILL LEARN

- Summarize what reinforcement learning does reliably in 2026 and where it still fails, separating established results from reported claims and from hopes.
- Explain why agents overfit to their training environments, and evaluate generalization with held-out levels and contexts.
- Describe how sequence models and pretrained foundation models are used for decision making, and what reinforcement learning contributes when pretraining does most of the work.
- Explain meta-reinforcement learning, in-context learning of RL algorithms, and the loss of plasticity in continual learning, and test remedies in a small experiment.
- Describe the complexity measures of RL theory with function approximation, and the gap between that theory and deep RL practice (●●●).
- Analyze what made reinforcement learning work in scientific and engineering applications.
- Turn an open problem into a research question with a small, decisive experiment.

The first twenty-three chapters of this book describe methods that work: their derivations, their guarantees when there are any, and the experiments that show what they do. This last chapter looks outward. It is a guided tour of the questions that the field is working on in 2026, written for a reader who has finished the book (or a good part of it) and wants to know where the edges are and how to start working on them.

A tour of a fast-moving field has to be careful. Research papers report results under particular conditions, and some striking claims have not been reproduced; some debates are open. In this chapter, "it is known" means proved or reproduced many times, "reported" means claimed in a paper (with a citation) and not independently established, and the open problems are open. Where the book's own small experiments illustrate a point, they are only that: illustrations on toy problems, which can make an idea concrete but cannot settle a question about large systems.

Figure 24.1 shows the plan. Section 24.1 takes stock of what reinforcement learning can and cannot do. Four sections then look at frontiers that grow directly out of the book: generalization across environments (Section 24.2), foundation models used as policies (Section 24.3), learning to learn (Section 24.4), and learning that never stops (Section 24.5). Section 24.6 (●●●) surveys the theory of RL with function approximation. Section 24.7 asks what made the scientific and engineering successes of RL work. The chapter ends with a curated list of open problems (Section 24.8) and advice on doing research (Section 24.9). Language models, a frontier of their own, have Chapter 22; the questions that the reward hypothesis raises were discussed in Chapter 1

and are not repeated here.

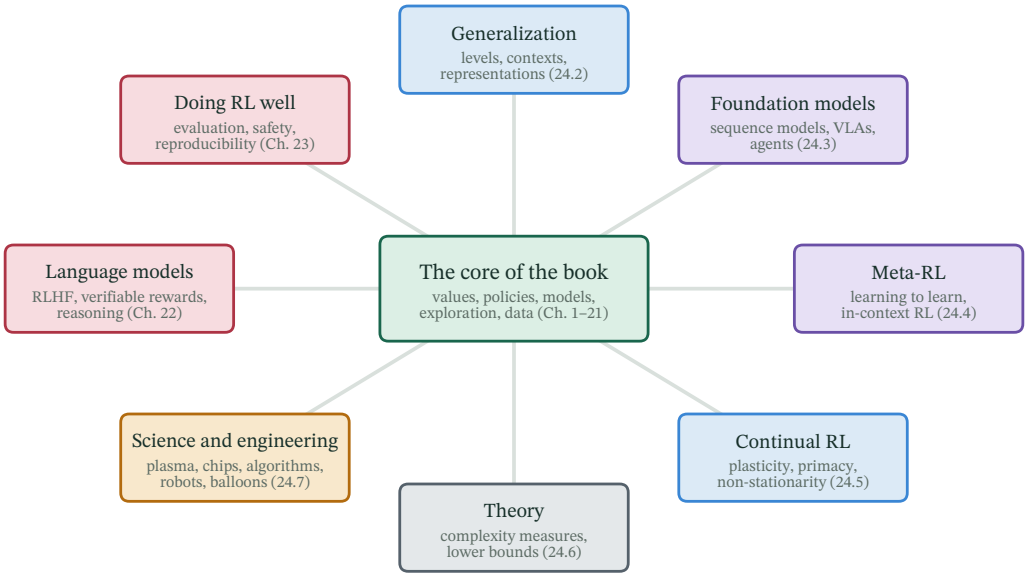


Figure 24.1. A map of the frontiers discussed in this chapter, around the core methods of the book. Language models and the practice of RL have their own chapters; the numbers in parentheses are the sections of this chapter.

24.1 Where the field stands

A balanced picture starts with what works. Reinforcement learning reliably produces superhuman play in games that can be simulated cheaply and exactly, from backgammon to Go, chess and poker, mostly through self-play combined with search (Section 15.7, Section 21.3.2). It produces controllers for robots, drones, racing cars and physical systems when a good simulator exists and the gap between simulation and reality can be bridged (Section 23.8). It is a standard part of training language models, both from human preferences and from automatically checkable rewards (Chapter 22). And it solves well-posed combinatorial search problems when the objective can be computed exactly, as in the discovery of new algorithms (Section 24.7).

These successes share a pattern. Each had a way to generate enormous amounts of experience cheaply (a simulator, a game engine, a checker), a reward that measured what mattered, and the computation to use them. Where these ingredients are missing, reinforcement learning remains hard.

- *Sample efficiency.* Deep RL agents often need millions to billions of interactions. Learning directly in the physical world, where each trial costs time, money or risk, is rarely practical without a simulator, demonstrations or prior data.
- *Reliability.* Results depend on random seeds, implementation details and hyperparameters to a degree that is unusual in machine learning (Henderson et al. 2018), and published comparisons have often been based on too few runs to be trusted (R. Agarwal et al. 2021). Chapter 23 treats this in detail.

- *Generalization.* Agents trained in one environment often fail in slightly different ones (Section 24.2).
- *Reward specification.* Optimizing a proxy reward hard enough exposes its flaws (Section 22.8, Section 23.6).
- *Safety and guarantees.* Methods with formal safety guarantees exist for restricted settings (Section 23.7), but deep RL systems come with few guarantees about their behavior outside the conditions they were trained in.

Two broad views of the way forward are worth knowing, because they shape what people work on. Sutton's essay *The Bitter Lesson* (Sutton 2019) argues from the history of artificial intelligence that general methods that scale with computation, above all search and learning, eventually beat methods that build in human knowledge. The opposing emphasis holds that the right inductive biases (models, abstractions, structure in the representation) are what make learning efficient enough to be practical. The two are less opposed than they sound, since a well-chosen bias is often what makes a general method scale; our generalization experiment below shows both forces at once.

24.2 Generalization

In supervised learning, nobody would report the accuracy of a classifier on the images it was trained on. In reinforcement learning this was long the standard protocol: the classic Atari benchmark trains and evaluates the agent on the same games, with the same levels. An agent can then score well by memorizing what to do in situations it has seen, without learning anything that would help in a new one.

Studies with *procedurally generated* environments made the problem measurable. Such environments generate each level (a maze, a platform game screen, a dungeon) from a random seed, so an agent can be trained on one set of levels and tested on levels it has never seen. Zhang et al. (2018) documented that deep RL agents can overfit their training levels this way. With the CoinRun game, Cobbe et al. (2019) found that agents overfit even to surprisingly large training sets of levels, and that regularization techniques from supervised learning, such as L_2 penalties, dropout and data augmentation, reduce the gap. The Procgen benchmark (Cobbe et al. 2020) made this a standard: sixteen procedurally generated games with a protocol that trains on a fixed number of levels and tests on the full distribution. Agents trained on few levels show large *generalization gaps* between training and test performance, the gap shrinks as the number of training levels grows, and the authors reported that larger network architectures improved both sample efficiency and generalization.

The formal setting. A clean way to state the problem is the *contextual MDP* (Hallak et al. 2015).

Definition 24.1 Contextual MDP and generalization gap

A *contextual MDP* is a set of MDPs $\{M_c : c \in \mathcal{C}\}$ that share their state and action spaces, together with a distribution $\mathcal{P}$ over *contexts* c. Each episode draws a context $c \sim \mathcal{P}$ and runs in M_c. A policy

π is trained on a finite set of contexts $C_{\text{train}} = \{c_1, \dots, c_n\}$, and its *generalization gap* is

$$\frac{1}{n}\sum_{i=1}^{n} J_{M_{c_i}}(\pi) - \mathbb{E}_{c\sim\mathcal{P}}\big[J_{M_c}(\pi)\big],$$

the difference between its average return on the training contexts and its expected return on a new context.

A level of a procedurally generated game is a context; so is the friction of a simulated floor, the layout of a warehouse, or the user of a recommender system. If the agent can observe the context (or infer it from its observations), the problem is a large MDP; if it cannot, it is a POMDP with the context as a hidden state (Chapter 19), which is the view that meta-RL takes (Section 24.4). The survey by Kirk et al. (2023) organizes the field around this formalism.

24.2.1 An experiment with procedurally generated grids

Lab 24.1 builds a miniature version of this protocol. A level is a 6×6 grid with walls placed at random (each cell is a wall with probability 0.25), a start and a goal at least three moves apart, and a check by breadth-first search that the goal is reachable; Figure 24.2 shows four levels. The agent moves up, down, left or right, earns $+1$ on reaching the goal, and is evaluated by whether its greedy policy reaches the goal within 18 moves. It is trained by fitted Q-iteration (Algorithm 9.6) with a two-layer network on every transition of n training levels, with the same amount of computation for every n, and tested on 200 levels it has never seen.

We compare two observations. The *global* observation shows the whole grid as three one-hot images (walls, goal, agent). The *egocentric* observation shows the walls and the goal in a 5×5 window centered on the agent, plus the goal's position relative to the agent. Both contain enough information to act optimally in this task; they differ in what the network must learn to treat as "the same situation".

The results (Figure 24.2) show three effects. First, with few training levels both agents master their training levels and fail on new ones: with one training level, the test success is 2% for the global observation and 5.5% for the egocentric one, while a random policy succeeds 12% of the time. A deterministic policy that has memorized its levels can be worse than random on new ones, because it repeats the same wrong move forever. Second, more levels close the gap for the egocentric agent: with 256 levels its test success is 85%, equal to its training success. Third, the global agent barely improves (from 2% to 20% with 256 levels) and, with the fixed computation budget, no longer even fits its training levels. The same data, presented differently, generalizes or does not.

24.2.2 What helps

The remedies studied in the literature fall into four families, and the experiment illustrates two of them.

More diverse training data. More levels, more contexts, randomized physical parameters (Section 23.8). This is the most reliable remedy when it is affordable.

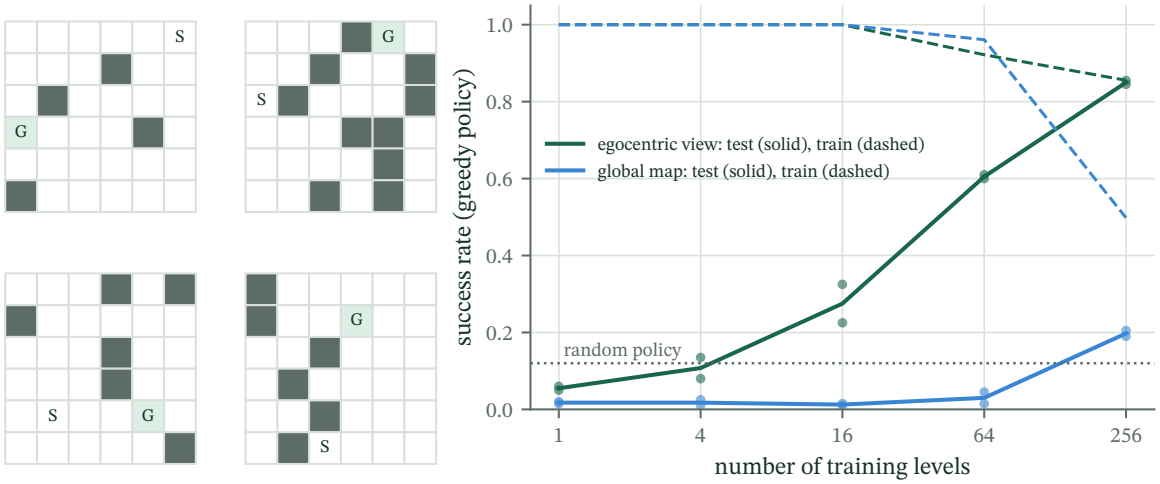


Figure 24.2. Generalization across procedurally generated levels. Left: four levels (S start, G goal, dark cells are walls). Right: success of the greedy policy of fitted Q-iteration on its training levels (dashed) and on 200 new levels (solid), as the number of training levels grows; lines are means over two seeds, dots individual seeds. The dotted line is a policy that moves at random. Only the egocentric observation turns more levels into generalization.

Inductive biases. Architectures and observation designs that make the right situations look alike: convolutional networks for images, egocentric views, object-centric or relational representations. The egocentric observation above builds in translation invariance; a convolutional network would learn it more slowly from the global map.

Data augmentation and regularization. Training on transformed copies of observations (random crops, shifts, color changes) improves both data efficiency and generalization in pixel-based control (Laskin et al. 2020; Kostrikov et al. 2021). In the grid task, adding the eight rotations and reflections of each training level is an augmentation (Exercise 24.2).

Representation learning. Auxiliary objectives that shape the representation (predicting future observations, contrasting augmented views, matching states that behave alike), so that the policy is learned on features that ignore irrelevant variation.

What is specific to reinforcement learning is that the agent also chooses its data. Exploration decides which situations it has seen, and a value function trained by bootstrapping can overfit in ways that a policy does not. How these interact with generalization is not well understood; see the open problems.

CHECKPOINT

1. In the contextual-MDP view, what is a "level" of a procedurally generated game?
2. Why can a deterministic policy that memorized its training levels do worse than a random policy on a new level?
3. Name two ways to reduce a generalization gap without collecting more levels.

Answers.

1. A context c: one MDP M_c drawn from the distribution of levels, sharing states and actions with the others.
2. On a new level its memorized moves can lead into a wall or a loop, and a deterministic policy repeats the same move forever, whereas a random policy eventually wanders to the goal on small levels.
3. Build invariances into the observation or architecture (for example an egocentric view or convolutions), and augment the data with transformations that preserve the task (rotations, reflections, crops); representation learning and regularization are other answers.

PITFALL

Testing on the training environments measures memorization. When you evaluate an agent, say which levels, seeds, contexts or parameter settings it was trained on and which it is tested on, and report both numbers. "The agent solves the task" is not a claim about generalization unless the test conditions were held out.

24.3 Foundation models and decision making

The largest change in the field since the mid-2010s is that many decision-making systems now start from a model pretrained on enormous amounts of data, rather than from a random network. This section describes three ways of using such models and asks what reinforcement learning still contributes.

Sequence models as policies. The *Decision Transformer* (L. Chen et al. 2021) treats offline RL as sequence modeling: a transformer reads a sequence of (return to go, state, action) triples and is trained to predict the actions; at test time it is conditioned on a high desired return (Chapter 17 discusses when this works). *Gato* (Reed et al. 2022) went further and trained one transformer, with one set of weights, by supervised learning on hundreds of tasks spanning Atari games, image captioning, dialogue and the control of a real robot arm. These models are trained by imitation, not by reinforcement learning; they show how much of decision making can be learned by predicting what a competent agent did.

Vision-language-action models. For robots, *vision-language-action* (VLA) models fine-tune a pretrained vision-language model to output robot actions, written as tokens like words. RT-2 (Brohan, Brown, Carbajal, Chebotar, Chen, et al. 2023) trained such a model jointly on web data and robot trajectories and reported better generalization to objects and instructions that were absent from the robot data, inherited from the web-scale pretraining. OpenVLA (Kim et al. 2024) is an open model of this kind trained on a large collection of robot demonstrations. These systems too are trained mostly by imitation; the robot data are demonstrations.

Language models as agents. A language model can serve as the high-level planner of an agent. In SayCan (Ahn et al. 2022) a language model proposes which skill a robot should execute next to follow an instruction, and value functions of the skills, learned with RL, estimate whether each skill can succeed in the current situation; the product of the two chooses the skill. Language-model agents that interleave reasoning with tool calls (Yao, Zhao, et al. 2023) are now common, and training them with reinforcement learning is an open frontier (Section 22.6).

What reinforcement learning contributes. When pretraining provides most of the competence, RL still does four things that imitation cannot. It improves behavior *beyond* the data, using a reward instead of a demonstration, as in the fine-tuning of language models (Chapter 22). It

supplies *grounding*: value functions that say what is achievable from here, as in SayCan. It trains *closed-loop* behavior that recovers from its own mistakes, where imitation compounds them (Theorem 18.1). And it provides the framework for agents that improve with experience after deployment. Whether large pretrained models plus RL fine-tuning will be enough for general-purpose agents, or whether new ingredients are needed, is one of the large open questions of the field.

> **REMARK**
>
> Most of the systems in this section were trained with supervised learning on demonstrations, sometimes followed by RL fine-tuning. When reading about "foundation models for decision making", it is worth checking which part of the training used rewards at all.

24.4 Meta-RL and in-context RL

People learn new games in minutes because they bring experience from many previous games. *Meta-reinforcement learning* asks for the same: an agent trained on a distribution of tasks that learns each new task from the distribution quickly. The output of meta-training is not a policy for one task but a *learning procedure*.

The problem is cleanest as a POMDP. Draw a task $c \sim \mathcal{P}$, hide it from the agent, and let the agent interact with M_c for a fixed budget of episodes; the objective is the expected total return over the whole budget, including the early, uninformed episodes. The hidden task is part of the state, and the agent's history is all it knows about it. The best possible meta-learner is therefore the *Bayes-optimal* policy of this POMDP, which explores exactly as much as the remaining budget justifies (Exercise 24.4).

Learning algorithms in the weights of a recurrent network. RL^2 (Duan et al. 2016) and the closely related work of J. X. Wang et al. (2016) train a recurrent network whose input at each step includes the previous action, the reward and a flag marking episode boundaries, and whose hidden state is carried across episodes of the same task. An ordinary RL algorithm trains the network across many tasks (the "slow" outer loop). Within a task, the network's activations implement a learning algorithm (the "fast" inner loop) that never changes a weight: a recurrent network trained on bandit problems, for example, learns to balance exploration and exploitation in its hidden state.

Learning an initialization. MAML (Finn et al. 2017) instead meta-learns the initial parameters of a network so that a few gradient steps on a new task produce a good policy. Methods with explicit task inference, such as PEARL (Rakelly et al. 2019) and VariBAD (Zintgraf et al. 2020), learn a probabilistic latent variable that summarizes what the agent has learned about the task, and condition the policy on it; VariBAD makes the connection to Bayes-adaptive MDPs explicit. The survey by Beck et al. (2023) organizes the area.

In-context reinforcement learning with transformers. Large sequence models can learn to learn in context. *Algorithm distillation* (Laskin et al. 2023) trains a transformer to predict the actions of an RL algorithm from long histories of its learning across many tasks; the trained transformer then improves its own behavior on a new task, in context, without any update of its weights.

A transformer pretrained to predict optimal actions from in-context interaction data (Lee et al. 2023) was shown to explore new bandit and MDP tasks in a way that resembles posterior sampling (Algorithm 16.2).

Open-ended learning. At a larger scale, DeepMind's XLand work trained agents on a vast space of procedurally generated 3D games, with a curriculum that kept tasks at the edge of the agents' ability, and reported agents that could handle held-out games zero-shot (Open Ended Learning Team et al. 2021). Its successor, AdA (Adaptive Agent Team et al. 2023), combined meta-RL across a still larger task space with a large transformer memory and an automated curriculum, and reported adaptation to new tasks within a few episodes, on a timescale comparable to human players.

HISTORICAL NOTE

The idea that a learning system can improve its own learning procedure is older than deep learning: Jürgen Schmidhuber's work on self-referential learning in 1987 (Schmidhuber 1987) and the learning-to-learn research of the 1990s (Thrun and Pratt 1998) explored it for neural networks and other learners. RL^2 and the work of Wang and colleagues in 2016 revived it for reinforcement learning, and showed that a recurrent network trained by one RL algorithm can come to implement another.

Discovering the learning rule itself. One can also meta-learn the update rule of an RL algorithm. Oh et al. (2020) meta-learned a policy-update rule on simple environments and found that it transferred to Atari games. Building on this line, Oh et al. (2025) reported a rule meta-learned from the experience of many agents across many environments that outperformed hand-designed rules on the Atari benchmark and performed well on benchmarks not used during discovery. The usual caution applies with special force here: a rule meta-learned on one distribution of environments may encode that distribution, and testing on genuinely different environments is the point.

PITFALL

Meta-RL overfits too, at the level of tasks. An agent meta-trained on a narrow distribution of tasks learns to adapt within that distribution, and the "fast learning" may simply be the recognition of which of the training tasks it is facing. Evaluate on held-out tasks, and report the meta-training budget separately from the adaptation budget.

24.5 Continual RL and plasticity

Most of this book assumes a fixed problem and an agent that learns until it is good enough. Many real problems do not hold still: users change, markets change, a robot's joints wear out, and an agent deployed for years should keep learning. *Continual reinforcement learning* studies agents that learn without end, on problems that change.

The definition problem. The field has struggled to define its own subject. The review by Khetarpal et al. (2022) surveys the many existing formulations (sequences of tasks, non-

stationary MDPs, lifelong learning) and the desiderata, such as remembering old tasks, transferring to new ones and adapting quickly. Abel et al. (2023) propose a precise definition, in which continual RL is the setting where the best agents never stop learning; they argue that the standard view of learning as finding a solution and stopping does not fit such problems. There is no consensus yet on benchmarks or metrics (Exercise 24.5).

Non-stationarity is not only an external problem. Inside every value-based agent, the targets move as the value function changes, and the data distribution moves as the policy changes. A network trained by deep RL is a continual learner whether or not its environment changes.

Loss of plasticity. A striking finding of recent years is that neural networks trained on a long sequence of tasks gradually lose the ability to learn new ones. Dohare et al. (2024) demonstrated this *loss of plasticity* systematically, in supervised continual learning (for example sequences of classification tasks built from ImageNet) and in RL, with standard networks and optimizers: after enough tasks, the network learns each new task worse than a freshly initialized one would. Value-based RL shows related symptoms: bootstrapped training can make the learned features collapse to a low effective rank (Kumar et al. 2021), and the fraction of inactive ("dormant") units grows during training (Sokar et al. 2023). In continual Atari settings plasticity loss was also observed (Abbas et al. 2023). Lyle et al. (2023) found that no single statistic, such as the weight norm, the number of inactive units or the rank of the features, explains plasticity loss consistently across settings, connected it to the curvature of the loss landscape, and reported that layer normalization helps.

A related effect within a single task is the *primacy bias* (Nikishin et al. 2022): a deep RL agent overfits to its earliest experiences and fails to learn as much from later, better data. Periodically resetting the last layers of the networks while keeping the replay buffer improved sample efficiency in their experiments, and resets later allowed agents to make many more updates per environment step without degrading (D'Oro et al. 2023).

Remedies. Most proposed remedies inject fresh randomness or pull the network back toward its initial condition. *Resets* re-initialize the whole network or some layers. *Shrink-and-perturb* (Ash and Adams 2020) multiplies the weights by a factor below one and adds small noise. *Continual backpropagation* (Dohare et al. 2024) re-initializes a small fraction of the least useful units at every step. Others regularize toward the initial parameters, normalize layers, or change the activation function. Each trades some loss of what was learned for the ability to keep learning.

24.5.1 An experiment with changing targets

Lab 24.1 reproduces the phenomenon in miniature. A network with two hidden layers of 64 ReLU units is trained, for 300 Adam steps each, on 40 regression tasks in a row. All tasks use the same 128 inputs, and each task draws fresh random targets with a random offset of $+10$ or -10, a stand-in for the moving targets of value-based RL. After each task we record the training error reached, divided by the variance of the targets, and the fraction of *dead* units, which output zero on every input and so receive no gradient. Figure 24.3 compares three strategies.

At first, the network that keeps training is the best of the three: on tasks 2 to 6 its median normalized error is 0.035, against 0.41 for a network reset before every task, because what it learned

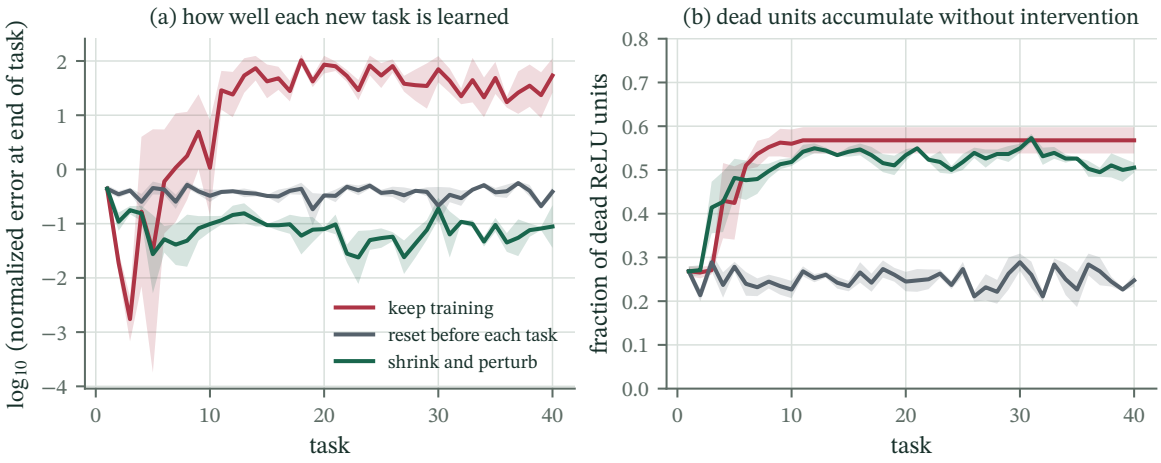


Figure 24.3. Loss of plasticity on a sequence of 40 regression tasks with changing targets (three seeds; lines are means, of the logarithm in panel a, and bands are ±1 standard error). (a) The error reached on each new task. A network that keeps training is the best at first, then loses the ability to fit new targets; one that is reset before each task never degrades; shrink-and-perturb keeps learning and keeps some benefit of experience. (b) Without intervention, more than half of the units die.

about the inputs carries over. Then it collapses. By tasks 36 to 40 its median error is 25, worse than predicting a constant, more than half of its units are dead (up from 27% after the first task), and the effective rank of its last hidden layer has fallen to zero. The reset network has the same error, about 0.41, at every task. Shrink-and-perturb keeps learning (median error 0.08 on the last tasks) although half of its units are dead at the end of each task, which illustrates that simple statistics do not tell the whole story. In this setup, the changing offset is the main culprit: without it, the continually trained network keeps fitting new tasks (Exercise 24.6).

CHECKPOINT

1. Why is every deep value-based agent a continual learner, even in a fixed environment?
2. In Figure 24.3, why is the continually trained network better than the reset network on the first few tasks?
3. What does shrink-and-perturb do at a task switch?

Answers.

1. Its regression targets change as its own value estimates change, and its data change as its policy changes.
2. All tasks share the same inputs, so the features learned on earlier tasks help to fit the new targets; a reset network starts from scratch every time.
3. It multiplies every weight by a factor below one (here 0.8) and adds small Gaussian noise, which keeps part of what was learned while restoring some of the randomness of a fresh initialization.

RESEARCH NOTE

Loss of plasticity is now well documented, but its causes are not settled, and the remedies are heuristics that trade memory for adaptability. Open questions: Which mechanisms drive plasticity loss in large RL agents, and are they the same as in supervised continual learning? Can an agent keep learning for an unbounded time without forgetting everything it has learned? What are the right metrics and benchmarks for continual RL?

24.6 Theory frontiers ●●●

The theory of the earlier chapters is sharpest for tabular problems, where sample complexity and regret scale with the numbers of states and actions (Theorem 16.3). Real problems have too many states for that to be meaningful, and the central theoretical question is: *which structural assumptions on a problem and a function class make reinforcement learning with function approximation statistically tractable*, that is, solvable with a number of samples polynomial in the horizon, the number of actions and a measure of the complexity of the class, but not in the number of states?

Realizability is not enough. The natural hope is that it suffices for the optimal value function to lie in the class, for instance to be linear in known features. It does not. Du et al. (2020) showed that if Q^* is only approximately linear, finding a near-optimal policy can require a number of samples exponential in the horizon, unless the approximation error is very small. Weisz et al. (2021) showed that even exact linear realizability of Q^*, with a simulator that can be queried at any state, allows exponential lower bounds, exponential in the smaller of the feature dimension and the horizon. Function approximation needs more structure than "the answer is in the class".

Complexity measures. Several measures identify such structure. The *Bellman rank* (Jiang et al. 2017) is the rank of a matrix of Bellman errors indexed by pairs of candidate value functions: when it is small, an algorithm (OLIVE) finds a near-optimal policy with polynomially many samples. The *eluder dimension* (Russo and Van Roy 2013) measures how many observations can each be surprising given the ones before, and governs optimistic exploration. The *Bellman eluder dimension* (C. Jin et al. 2021) combines the two ideas and covers tabular MDPs, linear MDPs (Chapter 16), problems of low Bellman rank and problems of low eluder dimension within one analysis; *bilinear classes* (Du et al. 2021) are another unifying framework. The *decision-estimation coefficient* (Foster et al. 2021) goes further and gives both lower and upper bounds on the achievable regret for general interactive decision making, in terms of one quantity that balances the regret of a decision against the information it reveals; the lecture notes of Foster and Rakhlin (2023) present this theory in a single framework.

Computation. These results are statistical: they count samples, not operations. Several of the algorithms that achieve them, including OLIVE, maintain sets of candidate functions and are not computationally efficient. Which statistically tractable classes are also computationally tractable is an active question, and the two are known to differ: unless NP = RP, no polynomial-time algorithm solves every deterministic MDP with a few actions whose optimal value functions are linear in known features, although polynomially many samples suffice (Kane et al. 2022).

Theory and practice. The gap between this theory and deep RL practice remains wide. Practical

agents use replay buffers, target networks, bootstrapping with nonlinear networks and extensive tuning; none of the algorithms with the guarantees above is what practitioners run, and none of the practical algorithms has guarantees of this kind. Bridging the two, by analyses of the algorithms that are used or by practical algorithms derived from theory, is one of the open problems below. The book draft of Agarwal et al. (2022) is the best place to start reading.

24.7 Reinforcement learning for science and engineering

Some of the most convincing evidence that reinforcement learning works comes from outside games and benchmarks. We describe a few prominent results, as reported in their papers, and then ask what they have in common.

Plasma control. Degrave et al. (2022) trained a deep RL controller for the magnetic coils of the TCV tokamak, an experimental fusion device in Lausanne, entirely in a simulator, and deployed it on the real machine without further training. It held plasmas in a range of shapes, including configurations with two separate plasma "droplets" at once. The reward specified the target shape, and a single learned controller replaced a set of hand-designed feedback loops.

Chip design (and a controversy). Mirhoseini et al. (2021) framed the placement of large circuit blocks on a chip as a sequential decision problem and reported that an RL agent with a graph neural network produced placements comparable or superior to those of human experts in under six hours. The claims were disputed. An independent assessment (Cheng et al. 2023) reported that, in its experiments, the method did not outperform simulated annealing or commercial placement tools; the original authors replied that the assessment departed from their method, for example by omitting pretraining and using less compute (Goldie et al. 2024). The episode is a reminder that strong baselines, open code and agreed benchmarks matter as much in applications as in research.

Discovering algorithms. AlphaTensor (Fawzi et al. 2022) turned the search for fast matrix multiplication algorithms into a single-player game over tensor decompositions, played by an AlphaZero-style agent (Section 15.7); among its findings was an algorithm that multiplies 4×4 matrices with entries modulo 2 using 47 multiplications, fewer than the 49 of applying Strassen's algorithm twice. AlphaDev (Mankowitz et al. 2023) searched over assembly-language programs and found sorting routines for short sequences that were faster than the human-written ones; they were added to the C++ standard library of the LLVM project. (FunSearch (Romera-Paredes et al. 2024), which found new constructions in combinatorics, used evolutionary search with a language model rather than reinforcement learning.)

Robots, drones, balloons and cars. A quadruped controller trained in simulation walked robustly over rough natural terrain (Lee et al. 2020); a robot hand trained in simulation with randomized physics solved a Rubik's cube (OpenAI et al. 2019); a racing drone trained in simulation beat human champions in head-to-head races (Kaufmann et al. 2023); a controller trained in simulation steered stratospheric balloons, and in a 39-day experiment over the Pacific Ocean kept them near their stations better than the controller previously in use (Bellemare et al. 2020); and an agent beat top human players of the racing simulator Gran Turismo (Wurman et al. 2022).

Table 24.1. What the applications have in common: cheap experience, a measurable objective, and a way across any gap between training and deployment.

Application	Experience came from	Reward	What made transfer work
tokamak plasma	physics simulator	match target shape	accurate simulator, robust policy
matrix multiplication	exact game	fewer multiplications	the game is the problem
sorting routines	program execution	correct and fast	exact checks of correctness
legged robots, drones	simulator	task progress	randomized and learned noise models
balloons	weather simulation	stay near the station	historical weather data, simulation

Table 24.1 summarizes the pattern. In every case experience was cheap: a simulator good enough to train in, or an exact game in which the problem itself is the environment. The objective was measurable, and checked automatically. Where the agent had to leave the simulator, its designers bridged the gap by randomizing the simulation, modeling the noise of the real system, or both (Section 23.8). And the work was done by teams that combined RL expertise with deep knowledge of the domain, which shaped the observations, the action spaces and the rewards. Problems without a good simulator or an automatic objective, such as most medical, educational and economic decisions, remain far harder, and the methods of Chapter 17 are often the only option.

24.8 Open problems

This section lists open problems that the author considers important and tractable in 2026. Each paragraph says why the problem matters, what is known, and where to start reading. The list is a personal selection, not a consensus, and some problems will look different in a few years.

1. Sample efficiency in the real world. Agents that need millions of trials cannot learn on real robots, with real patients or in real markets. Model-based methods (Section 15.6), offline pretraining (Section 17.1), demonstrations and better exploration all reduce the cost, but no method yet learns new physical skills with the efficiency of an animal. Start with Chapter 15 and the work on high replay ratios and resets (D'Oro et al. 2023).

2. Generalization and robustness. Agents fail under small changes of their environment (Section 24.2). We lack a theory of when RL generalizes, benchmarks whose test distribution reflects real deployment, and methods that reliably generalize beyond the training distribution. Start with Kirk et al. (2023) and Procgen (Cobbe et al. 2020).

3. Exploration in large spaces. The provable exploration methods of Chapter 16 work for small or structured spaces; intrinsic motivation (Section 16.7) works on some hard games but not reliably.

Exploring in the space of programs, proofs or chains of thought, where a language model is the policy, is largely unsolved (Section 22.6).

4. Credit assignment over long horizons. Rewards that arrive thousands of steps after the decisions that caused them are hard to learn from, whether the steps are motor commands or tokens of a long derivation. Temporal abstraction (Chapter 20), learned models and process rewards (Section 22.9) help in particular cases; a general solution is missing.

5. Reward specification and alignment. Every reward is a proxy, and strong optimizers exploit proxies (Section 22.8). How to learn rewards from people reliably, how to tell when a policy is exploiting its reward, and how to supervise systems that are better than their supervisors at a task are open, and growing in importance. Start with Skalse et al. (2022), Casper et al. (2023) and Section 18.7.

6. Learning from fixed data and evaluating without deployment. Offline RL (Chapter 17) promises to use logged data from medicine, recommendation or robotics, but choosing hyperparameters and estimating the value of a policy without deploying it remain unreliable (Section 17.3). Methods with guarantees (Section 17.7.2) are often too conservative in practice.

7. Continual learning. Agents that keep learning lose plasticity, forget, or both (Section 24.5). We need agreed definitions, metrics and benchmarks, an understanding of the mechanisms, and methods that keep learning indefinitely. Start with Abel et al. (2023) and Dohare et al. (2024).

8. Safe exploration and guarantees. An agent learning in the world must avoid catastrophic actions while it does not yet know which actions are catastrophic. Constrained formulations (Section 23.7) and shielding work when the dangers can be specified in advance; guarantees for deep RL agents in open environments are mostly absent.

9. Theory that explains deep RL. Why do deep value-based methods work at all, given the deadly triad (Section 9.8)? Which properties of practical algorithms (replay, target networks, normalization) are essential, and why? Analyses of the algorithms people actually use would guide practice (Section 24.6).

10. World models good enough to plan with. Learned models make errors that compound over long rollouts and that planners exploit (Lemma 15.1). Models that know where they are wrong, and planners that respect that, would unlock sample-efficient learning; video-prediction and latent world models (Section 15.6) are the current approaches.

11. Learning with other agents, including people. Multi-agent RL can find equilibria of competitive games, but cooperation with humans and other learners, equilibrium selection in general-sum games and ad hoc teamwork are far from solved (Chapter 21).

12. Discovering abstractions. Options, skills and goals make long problems tractable (Chapter 20), but agents rarely discover useful abstractions by themselves at scale. Whether pretrained models can supply them, and how an agent should build its own, are open.

13. Scaling laws for reinforcement learning. Supervised learning has well-studied relations between compute, data, model size and performance. For RL, how performance scales with environment interactions, model size and computation, and how to allocate a budget among them,

is much less understood, both in classic control and in the training of reasoning models (Section 22.6).

14. What reinforcement learning adds to pretrained models. When a pretrained model already contains most of the competence, does RL fine-tuning create new abilities or mainly sharpen existing ones (Chapter 22)? How should agents built on language models be trained to act over many steps, with tools and other agents? The answers will decide much of how RL is used in the coming years.

24.9 How to do RL research

Research is a craft, and like any craft it is learned by doing it, preferably next to someone who does it well. A few habits make a large difference, and reinforcement learning has some pitfalls of its own.

Reading papers. Read in passes (Keshav 2007): first the title, abstract, figures and conclusions, to decide whether the paper matters to you; then the paper itself, skipping proofs; and only for the papers you will build on, everything, until you could re-derive the main results or re-implement the method. In RL, read the experimental section with care. How many seeds were run, and is the variability reported (Section 23.4)? Were the baselines tuned with the same budget as the new method? Was the method evaluated on held-out environments, or on the ones used during development? Does the improvement exceed the differences that implementation details alone can cause?

Choosing problems. A good research problem is important (someone will care about the answer), tractable (you can make progress with your resources) and yours (you have an idea, a skill or a vantage point that others lack). Prefer questions to methods: "why does this algorithm fail in this situation?" usually leads further than "a new algorithm that beats the baselines". Places where theory and practice disagree, where a widely used heuristic has no explanation, or where a method works in one domain and fails in a similar one are rich sources of questions. The open problems above are deliberately broad; a research project takes one small, sharp piece of one of them.

Designing experiments. State the hypothesis before running the experiment, and design the experiment so that it could prove you wrong. Change one thing at a time, and include ablations that remove each component of a method. Give baselines the same tuning budget as your method. Run enough seeds to see the variability, and report confidence intervals rather than single runs (Chapter 23). Keep a log of every run, including the ones that failed.

Small-scale science. Many questions can be answered with small problems that run in minutes, where you can afford many seeds and even exact computation. This book's labs work this way: the Trail, the toy language model of Chapter 22 and the experiments of this chapter can be enumerated or run hundreds of times. The Behaviour Suite (Osband et al. 2020) collects small experiments that each probe one capability, such as exploration, memory or credit assignment. Obando-Ceron and Castro (2021) revisited the large-scale study behind Rainbow on small environments, at a fraction of its cost, and argued that such experiments still yield valuable scientific

insights. A small experiment that isolates a mechanism is often more convincing than a large one that confounds several.

Writing. Make the claims match the evidence, no stronger. Say what you did not test. Report negative results when they are informative. Release code, configurations and seeds so that others can reproduce your numbers, and describe the computation you used.

> **KEY IDEA**
>
> A checklist for an RL research project: a sharp question; the simplest setting in which the question arises; a hypothesis that an experiment could refute; strong, equally tuned baselines; ablations; enough seeds, with confidence intervals; held-out test conditions; honest limitations; released code.

24.10 Summary

- Reinforcement learning works reliably when experience is cheap, the objective is measurable and computation is plentiful: in games, in simulation-trained control, in fine-tuning language models and in exact search problems. Sample efficiency, reliability, generalization, reward specification and guarantees remain the main limitations.
- Agents overfit to their training environments. Procedurally generated levels and contextual MDPs make generalization measurable; more diverse data, good inductive biases, augmentation and representation learning reduce the gap. In our grid experiment, the same number of training levels led to generalization with an egocentric observation and to almost none with a global one.
- Pretrained sequence models, vision-language-action models and language-model agents bring broad competence to decision making, mostly through imitation; reinforcement learning contributes improvement beyond the data, grounding, closed-loop recovery and learning from experience.
- Meta-RL trains learning procedures across task distributions, in the weights of recurrent networks, in initializations, in latent task variables or in the context of transformers; its ideal is the Bayes-optimal policy of the POMDP over tasks.
- Networks that keep learning lose plasticity: they learn new tasks worse than fresh networks. Resets, shrink-and-perturb and selective re-initialization restore it, at a cost.
- (●●●) Function approximation needs structure beyond realizability; Bellman rank, eluder dimensions, bilinear classes and the decision-estimation coefficient describe tractable classes, but a wide gap separates this theory from deep RL practice.
- Good research asks sharp questions, tests them in the simplest setting that contains them, and reports what was and was not shown.

This is the end of the book's main text. The appendices collect the mathematics, the stochastic-approximation results and the neural-network background used throughout, and a guide to the labs.

24.11 Further Reading

A curated reading list, by theme. Each entry is a good first paper on its topic.

Taking stock. Henderson et al. (2018) and R. Agarwal et al. (2021) on what RL results can and cannot be trusted; Sutton (2019) for one view of what drives long-term progress; the debate about rewards in Chapter 1.

Generalization. Kirk et al. (2023), the survey that organizes the area around contextual MDPs; Cobbe et al. (2019) and Cobbe et al. (2020) for the procedurally generated benchmarks that made the problem visible; Laskin et al. (2020) and Kostrikov et al. (2021) for augmentation.

Foundation models and decision making. L. Chen et al. (2021) and Reed et al. (2022) for sequence models as policies; Brohan, Brown, Carbajal, Chebotar, Chen, et al. (2023) for vision-language-action models; Ahn et al. (2022) for combining language models with learned value functions.

Meta-RL and in-context RL. Duan et al. (2016) and J. X. Wang et al. (2016) for learning to learn with recurrent networks; Finn et al. (2017) for MAML; Beck et al. (2023) for a survey; Laskin et al. (2023) for in-context RL; Adaptive Agent Team et al. (2023) for open-ended meta-RL at scale; Oh et al. (2025) for meta-learned update rules.

Continual RL and plasticity. Khetarpal et al. (2022) and Abel et al. (2023) on what continual RL is; Dohare et al. (2024), Lyle et al. (2023) and Nikishin et al. (2022) on plasticity and how to keep it.

Theory (●●●). Agarwal et al. (2022) for the foundations; Jiang et al. (2017), C. Jin et al. (2021) and Foster et al. (2021) for complexity measures; Du et al. (2020) and Weisz et al. (2021) for the lower bounds; Foster and Rakhlin (2023) for a unified treatment.

Applications. Degrave et al. (2022), Fawzi et al. (2022), Mankowitz et al. (2023), Kaufmann et al. (2023) and Bellemare et al. (2020); for the chip-design debate, Mirhoseini et al. (2021), Cheng et al. (2023) and Goldie et al. (2024), read together.

Doing research. Keshav (2007) on reading papers; Osband et al. (2020) and Obando-Ceron and Castro (2021) on small-scale experiments; Chapter 23 on experimental design and evaluation.

24.12 Exercises

Most of these exercises are open-ended. The solutions give model answers that cover the main points a good answer should make; they are not the only correct answers.

Exercise 24.1 ● Reading the generalization experiment

From the numbers of Section 24.2.1 (and Lab 24.1), compute the generalization gap of each agent with 1, 16 and 256 training levels. Explain why the agent with the global observation is worse than a random policy on new levels when it has few training levels, and why its training success falls at 256 levels.

Exercise 24.2 ● Augmentation in the grid world

In Lab 24.1, train the agent with the global observation on 16 and on 64 levels together with the eight rotations and reflections of each level (so the data are eight times larger), with the same amount of computation as before. Compare its test success with that of the agent trained on the original levels only, and with that of an agent trained on 256 fresh levels. What does augmentation buy here, and what does it cost?

Exercise 24.3 ●● Knowing the context

A contextual MDP has two contexts with probability $\frac{1}{2}$ each, and a single decision: in context c_1 the action left earns 1 and right earns 0, in context c_2 the reverse. (a) What is the expected return of the best policy that observes the context, and of the best policy that does not? (b) Now the agent faces the same context for two consecutive decisions and observes its first reward before the second decision, without observing the context. Find the best policy and its expected total return. (c) Relate (b) to meta-RL and to POMDPs.

Exercise 24.4 ●● A Bayes-optimal meta-learner

Tasks are two-armed bandits: in task A, arm 1 pays 1 with probability 0.9 and arm 2 with probability 0.1; in task B the arms are swapped. Each task has probability $\frac{1}{2}$, and the agent plays two rounds in the same task without knowing which. (a) Compute the expected total reward of always pulling arm 1, of the Bayes-optimal policy, and of an oracle that knows the task. (b) Explain why a recurrent network trained with RL^2 on this task distribution should learn the Bayes-optimal policy. (c) What changes if the horizon is 100 rounds?

Exercise 24.5 ●● Measuring continual learning

Propose a metric for continual reinforcement learning that rewards an agent for performing well on the current task, for learning new tasks quickly, and for not losing ability over time. Apply it to the three strategies of Figure 24.3. What would your metric reward that it should not, or fail to reward that it should?

Exercise 24.6 ● What causes plasticity loss here?

In Task 4 of Lab 24.1, (a) set the target offset to zero (`target_shift=0`) and (b) replace ReLU by tanh units (`activation='tanh'`), and compare the error of the continually trained network on tasks 2 to 6 and 36 to 40 with that of the reset network. What do the results suggest about the cause of plasticity loss in this experiment, and why should you be careful about generalizing from it?

Exercise 24.7 ●● Critique an evaluation

A paper proposes a new deep RL algorithm. It reports the mean return over 3 seeds on 5 Atari games chosen by the authors, compares with a baseline whose hyperparameters were taken from

an older paper while the new method's were tuned on the same 5 games, evaluates on the training environments, and claims "a new state of the art". List the problems with this evaluation, say how each could change the conclusion, and describe an evaluation that would support the claim.

Exercise 24.8 ●● Design a generalization benchmark

Design a benchmark for zero-shot generalization in a domain of your choice (for example robotic grasping, recommendation or traffic control). Specify the contexts, the training and test splits, what is held out and why, the metrics, the baselines, and one way in which an agent could score well on your benchmark without generalizing in the sense you intend.

Exercise 24.9 ●● Anatomy of an application

Choose one application from Section 24.7 and read its paper. Identify the environment and where the experience came from, the state, action and reward, what the system had to transfer to and how, the baselines, and the claims. Then name one change to the problem that would make reinforcement learning much harder, and explain why.

Exercise 24.10 ●●● Why realizability is not enough

Explain in your own words why knowing that Q^* is linear in known features does not make reinforcement learning sample-efficient, and what additional structure the Bellman rank or the Bellman eluder dimension assumes. Use a sketch of a hard example in your explanation, and say which of the assumptions a deep Q-network implicitly relies on.

Exercise 24.11 ●●● A research proposal

Choose one open problem from Section 24.8 and write a one-page research proposal: the specific question, why it matters, the closest prior work, a hypothesis, a small-scale experiment (runnable on a laptop) that could refute the hypothesis, the baselines and measurements, and what you would do next for each possible outcome.

Exercise 24.12 ●●● The bitter lesson and inductive biases

Write a short essay on the claim of Sutton (2019) that general methods which scale with computation eventually beat methods that build in knowledge. Use the generalization experiment of Section 24.2.1 as evidence on one side or the other, and discuss what it can and cannot show.

APPENDIX A

Mathematical Toolkit

This appendix collects the mathematics the book uses, with short examples from reinforcement learning. It is meant to be consulted rather than read straight through: when a chapter uses an idea you want to refresh, look it up here. The Foundations sections of the book need only Section A.1 and the first part of Section A.3; the Graduate and Research sections use the rest.

A.1 Probability

A.1.1 Random variables and expectations

A *random variable* X assigns a number to each outcome of a random experiment. If X takes finitely many values $x_1, \dots, x_n$ with probabilities $p_1, \dots, p_n$, its *expectation* (mean) is

$$\mathbb{E}[X] = \sum_{i=1}^{n} p_i \, x_i,$$

and for a variable with a density $p(x)$ it is $\mathbb{E}[X] = \int x \, p(x) \, dx$. More generally $\mathbb{E}[f(X)] = \sum_i p_i f(x_i)$ for any function f. Expectation is *linear*: $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$ for any random variables X, Y and constants a, b, whether or not X and Y are independent. Linearity is used constantly; for instance it lets us write the expected return as a sum of expected rewards, $\mathbb{E}[G_0] = \sum_t \gamma^t \mathbb{E}[r_t]$.

The *variance* measures spread around the mean:

$$\mathrm{Var}(X) = \mathbb{E}\big[(X - \mathbb{E}[X])^2\big] = \mathbb{E}[X^2] - \mathbb{E}[X]^2,$$

and the *standard deviation* is its square root. For independent X and Y, $\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$; in general $\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X,Y)$ with $\mathrm{Cov}(X,Y) = \mathbb{E}[(X - \mathbb{E}X)(Y - \mathbb{E}Y)]$.

Example A.1 The return of the Trail as a random variable

Under the uniform policy of the Trail with $\gamma = 0.9$ (Chapter 1), the return G_0 takes the values 1, 1.8 and 3.24 with probabilities $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{4}$. Its mean is $\mathbb{E}[G_0] = 0.5 + 0.45 + 0.81 = 1.76$, which is the value $V^\pi(\text{Start})$. Its second moment is $\mathbb{E}[G_0^2] = 0.5 + 0.81 + 2.6244 = 3.9344$, so $\mathrm{Var}(G_0) = 3.9344 - 1.76^2 = 0.8368$ and the standard deviation is about 0.915.

♦

An *indicator* $\mathbb{1}\{A\}$ is 1 if the event A happens and 0 otherwise. Its expectation is a probability: $\mathbb{E}[\mathbb{1}\{A\}] = \Pr(A)$. This small fact turns counting arguments into expectations, for example the expected number of times a bandit algorithm pulls a bad arm.

A.1.2 Conditional probability and Bayes' rule

The probability of A given that B happened is $\Pr(A \mid B) = \Pr(A \cap B)/\Pr(B)$ for $\Pr(B) > 0$. Rearranging gives *Bayes' rule*,

$$\Pr(A \mid B) = \frac{\Pr(B \mid A)\,\Pr(A)}{\Pr(B)},$$

which turns a model of how observations arise ($\Pr(B \mid A)$) into an inference about their cause. Thompson sampling (Chapter 2) and belief updates in partially observable problems (Chapter 19) are applications. The *law of total probability* says $\Pr(B) = \sum_i \Pr(B \mid A_i)\Pr(A_i)$ when the events A_i partition the outcomes.

A.1.3 Conditional expectation and the tower rule

The *conditional expectation* $\mathbb{E}[X \mid Y = y]$ is the mean of X when we know that $Y = y$. Viewed as a function of the random Y, $\mathbb{E}[X \mid Y]$ is itself a random variable, and the *tower rule* (law of total expectation) says

$$\mathbb{E}\big[\mathbb{E}[X \mid Y]\big] = \mathbb{E}[X]. \tag{A.1}$$

In words: averaging the conditional averages gives the overall average. The Bellman equation is the tower rule applied to the return. Conditioning on the first action and next state,

$$\begin{aligned} V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] &= \mathbb{E}_\pi\big[\, r_t + \gamma\, \mathbb{E}_\pi[G_{t+1} \mid s_{t+1}] \mid s_t = s\big] \\ &= \mathbb{E}_\pi\big[r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s\big], \end{aligned}$$

where the Markov property is what allows the inner expectation to depend on s_{t+1} alone. Two related facts: if X and Y are independent, $\mathbb{E}[X \mid Y] = \mathbb{E}[X]$; and anything known given Y can be taken out, $\mathbb{E}[f(Y)X \mid Y] = f(Y)\,\mathbb{E}[X \mid Y]$.

A.1.4 Common distributions

Table A.1. Distributions used in the book.

Distribution	Values	Mean	Variance	Where it appears
Bernoulli(p)	$\{0, 1\}$	p	$p(1-p)$	bandit rewards, success or failure
Categorical($p_1, \dots, p_K$)	$\{1, \dots, K\}$	—	—	softmax policies, discrete actions
Gaussian $\mathcal{N}(\mu, \sigma^2)$	$\mathbb{R}$	μ	σ^2	continuous actions, noise

Distribution	Values	Mean	Variance	Where it appears
Beta(α, β)	$[0, 1]$	$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$	posterior of a Bernoulli mean
Dirichlet$(\alpha_1, \dots, \alpha_K)$	probability vectors	$\frac{\alpha_k}{\sum_j \alpha_j}$	—	posterior of transition probabilities

The Beta and Dirichlet distributions are *conjugate priors*: if the prior on a Bernoulli mean is Beta(α, β) and we observe s successes and f failures, the posterior is Beta$(\alpha + s, \beta + f)$. The Dirichlet does the same for categorical outcomes, with one count per outcome. This is why posterior sampling for bandits and MDPs (Chapter 2 and Chapter 16) is cheap.

A.1.5 Averages: the law of large numbers and the central limit theorem

Let $X_1, X_2, \dots$ be independent copies of a random variable X with mean μ and variance σ^2, and let $\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$ be their average.

- The *law of large numbers* says that $\bar{X}_n \to \mu$ as $n \to \infty$ (with probability one). This is why Monte Carlo methods work.
- The average has mean μ and variance σ^2/n, so its *standard error* is $\sigma/\sqrt{n}$. Halving the error takes four times as many samples.
- The *central limit theorem* says that for large n the distribution of $\bar{X}_n$ is approximately Gaussian, $\mathcal{N}(\mu, \sigma^2/n)$. Hence $\bar{X}_n \pm 1.96\,\hat{\sigma}/\sqrt{n}$ is an approximate 95% confidence interval for μ, where $\hat{\sigma}$ is the sample standard deviation.

The normal approximation can be poor for small n or heavy-tailed returns; Appendix B gives inequalities that hold for every n, and Chapter 23 discusses bootstrap intervals.

A.1.6 Two identities used everywhere

Importance sampling. If p and q are distributions and $q(x) > 0$ wherever $p(x)f(x) \neq 0$, then

$$\mathbb{E}_{x\sim p}[f(x)] = \sum_x p(x)f(x) = \sum_x q(x)\frac{p(x)}{q(x)}f(x) = \mathbb{E}_{x\sim q}\Big[\frac{p(x)}{q(x)}f(x)\Big].$$

Samples from q, reweighted by the ratio p/q, estimate expectations under p. Off-policy learning (Chapter 5 and Chapter 17) applies this with p the target policy's trajectory distribution and q the behavior policy's.

The score function (log-derivative) identity. If p_θ is a family of distributions,

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta}[f(x)] = \sum_x f(x)\,\nabla_\theta p_\theta(x) = \sum_x f(x)\,p_\theta(x)\,\nabla_\theta \log p_\theta(x) = \mathbb{E}_{x\sim p_\theta}\big[f(x)\,\nabla_\theta \log p_\theta(x)\big],$$

using $\nabla p = p\,\nabla \log p$. With $f \equiv 1$ it gives $\mathbb{E}[\nabla_\theta \log p_\theta(x)] = 0$: the score has mean zero. These two lines are the core of the policy gradient theorem and of the fact that baselines do not bias it (Chapter 11).

A.2 Markov chains

A *Markov chain* on a finite set of states $\{1, \dots, n\}$ is described by a *transition matrix* P whose entry P_{ij} is the probability of moving from i to j; every row sums to one. A fixed policy turns an MDP into a Markov chain with matrix P^π. If d_t is the distribution of the state at time t, written as a row vector, then

$$d_{t+1}^\top = d_t^\top P, \qquad d_t^\top = d_0^\top P^t.$$

A distribution d with $d^\top P = d^\top$ is *stationary*: once the chain is distributed as d, it stays so. A chain is *irreducible* if every state can reach every other, and *aperiodic* if it does not cycle deterministically. For an irreducible, aperiodic finite chain, the stationary distribution d is unique, $d_t \to d$ from any start, and long-run averages converge: $\frac{1}{T}\sum_{t<T} f(s_t) \to \sum_s d(s) f(s)$ (the *ergodic theorem*). On-policy linear TD converges because it samples states from this distribution (Chapter 9).

Episodic tasks give *absorbing* chains: some states, once entered, are never left. If Q is the block of P among the non-absorbing states, the expected number of visits to j starting from i before absorption is the (i, j) entry of the *fundamental matrix* $(I-Q)^{-1}$. Discounting gives the analogous object for every chain: $(I - \gamma P)^{-1}$ counts *discounted* visits, and

$$d_\gamma^\top = (1-\gamma)\, d_0^\top (I - \gamma P)^{-1} = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t d_0^\top P^t$$

is the discounted visitation distribution of Definition 3.8.

A.3 Linear algebra

A.3.1 Vectors, norms and inner products

A value function on a finite state space is a vector $V \in \mathbb{R}^{|\mathcal{S}|}$, so the language of linear algebra fits naturally. The norms used in the book are

$$\|x\|_1 = \sum_i |x_i|, \qquad \|x\|_2 = \Big(\sum_i x_i^2\Big)^{1/2}, \qquad \|x\|_\infty = \max_i |x_i|, \qquad \|x\|_D = \Big(\sum_i d_i\, x_i^2\Big)^{1/2},$$

where the *weighted norm* $\|\cdot\|_D$ uses positive weights d_i, collected in the diagonal matrix $D = \mathrm{diag}(d)$. The max norm $\|\cdot\|_\infty$ is the natural norm for dynamic programming (Chapter 4); the weighted norm with the state distribution as weights is the natural norm for function approximation (Chapter 9). The inner product $\langle x, y\rangle = x^\top y$ satisfies the Cauchy–Schwarz inequality $|\langle x, y\rangle| \le \|x\|_2 \|y\|_2$, and Hölder's inequality $|\langle x, y\rangle| \le \|x\|_1 \|y\|_\infty$.

A matrix A has the *induced norm* $\|A\| = \max_{x \neq 0} \|Ax\|/\|x\|$. For the max norm, $\|A\|_\infty = \max_i \sum_j |A_{ij}|$, the largest absolute row sum. A stochastic matrix has $\|P\|_\infty = 1$: averaging cannot increase the largest entry. This is the one-line reason why $\|\gamma P^\pi V - \gamma P^\pi V'\|_\infty \le \gamma \|V - V'\|_\infty$.

A.3.2 Eigenvalues, definiteness and the Neumann series

An *eigenvalue* λ of a square matrix A satisfies $Ax = \lambda x$ for some $x \neq 0$; the *spectral radius* $\rho(A)$ is the largest $|\lambda|$, and $\rho(A) \leq \|A\|$ for every induced norm. A symmetric matrix has real eigenvalues and an orthonormal basis of eigenvectors. It is *positive definite* if $x^\top Ax > 0$ for all $x \neq 0$ (equivalently, all eigenvalues are positive), and *positive semidefinite* if $x^\top Ax \geq 0$. Covariance matrices and Fisher information matrices (Chapter 12) are positive semidefinite.

Proposition A.1 Neumann series

If $\|A\| < 1$ for some induced norm, then $I - A$ is invertible and

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k, \qquad \|(I - A)^{-1}\| \leq \frac{1}{1 - \|A\|}.$$

Proof. The partial sums $S_K = \sum_{k=0}^{K} A^k$ satisfy $\|S_L - S_K\| \leq \sum_{k=K+1}^{L} \|A\|^k$, so they form a Cauchy sequence and converge to some S with $\|S\| \leq \sum_k \|A\|^k = 1/(1-\|A\|)$. Since $(I-A)S_K = I - A^{K+1}$ and $\|A^{K+1}\| \leq \|A\|^{K+1} \to 0$, the limit satisfies $(I - A)S = I$, and likewise $S(I - A) = I$. ■

With $A = \gamma P^\pi$ and $\|\gamma P^\pi\|_\infty = \gamma < 1$, this proves that the Bellman equation $V = r^\pi + \gamma P^\pi V$ has the unique solution $V^\pi = (I - \gamma P^\pi)^{-1} r^\pi = \sum_k \gamma^k (P^\pi)^k r^\pi$: the expected reward at each future step, discounted and summed. This is the matrix form (3.9) of Chapter 3.

A.3.3 Projections and least squares

Given a feature matrix $\Phi \in \mathbb{R}^{n \times d}$ with linearly independent columns and positive weights $D = \mathrm{diag}(d)$, the vector in the column space of Φ closest to v in the norm $\|\cdot\|_D$ is Πv, with

$$\Pi = \Phi\,(\Phi^\top D \Phi)^{-1} \Phi^\top D.$$

To see it, minimize $\|\Phi w - v\|_D^2 = (\Phi w - v)^\top D (\Phi w - v)$ over w; setting the gradient $2\Phi^\top D(\Phi w - v)$ to zero gives the *normal equations* $\Phi^\top D \Phi\, w = \Phi^\top D v$. The projection is idempotent ($\Pi^2 = \Pi$), the error $v - \Pi v$ is D-orthogonal to every column of Φ, and so *Pythagoras* holds: $\|v\|_D^2 = \|\Pi v\|_D^2 + \|v - \Pi v\|_D^2$, which implies $\|\Pi v\|_D \leq \|v\|_D$ (projections never lengthen vectors). These facts are the backbone of the analysis of linear TD learning (Section 9.4).

A.4 Contraction mappings

Many algorithms in this book repeat one operation, $x_{k+1} = F(x_k)$, and hope to converge. The tool that guarantees it is the contraction mapping theorem.

A map F from a set X to itself is a *γ-contraction* in a norm $\|\cdot\|$ if, for some $\gamma \in [0, 1)$,

$$\|F(x) - F(y)\| \leq \gamma\, \|x - y\| \quad \text{for all } x, y \in X.$$

A point with $F(x^*) = x^*$ is a *fixed point*. We need X to be *complete*: every Cauchy sequence in X has a limit in X. Every closed subset of $\mathbb{R}^n$ is complete, which covers all value functions on finite state spaces.

Theorem A.2 Banach's fixed-point theorem

Let F be a γ-contraction on a complete space X. Then F has exactly one fixed point x^*, and for every starting point x_0 the iterates $x_{k+1} = F(x_k)$ satisfy

$$\|x_k - x^*\| \le \gamma^k \|x_0 - x^*\|, \qquad \|x_k - x^*\| \le \frac{\gamma^k}{1-\gamma}\|x_1 - x_0\|, \qquad \|x_k - x^*\| \le \frac{\gamma}{1-\gamma}\|x_k - x_{k-1}\|.$$

Proof. *The iterates form a Cauchy sequence.* By induction, $\|x_{k+1} - x_k\| \le \gamma^k \|x_1 - x_0\|$. For $m > k$, the triangle inequality gives

$$\|x_m - x_k\| \le \sum_{j=k}^{m-1} \|x_{j+1} - x_j\| \le \sum_{j=k}^{m-1} \gamma^j \|x_1 - x_0\| \le \frac{\gamma^k}{1-\gamma}\|x_1 - x_0\|,$$

which tends to zero as $k \to \infty$. By completeness the sequence has a limit x^*.

The limit is a fixed point. A contraction is continuous, so $F(x^*) = F(\lim_k x_k) = \lim_k F(x_k) = \lim_k x_{k+1} = x^*$.

It is the only one. If $F(y) = y$ as well, then $\|x^* - y\| = \|F(x^*) - F(y)\| \le \gamma \|x^* - y\|$, which forces $\|x^* - y\| = 0$ because $\gamma < 1$.

The error bounds. $\|x_k - x^*\| = \|F(x_{k-1}) - F(x^*)\| \le \gamma \|x_{k-1} - x^*\|$, and induction gives the first bound. Letting $m \to \infty$ in the Cauchy estimate gives the second. For the third, $\|x_k - x^*\| \le \|x_{k+1} - x_k\| + \|x_{k+1} - x^*\| \le \gamma \|x_k - x_{k-1}\| + \gamma \|x_k - x^*\|$; rearranging gives $(1-\gamma)\|x_k - x^*\| \le \gamma \|x_k - x_{k-1}\|$. ■

The third bound is the practical one: it tells you how far you are from the answer using only quantities you can compute, the change in the last iteration. The Bellman operators $\mathcal{T}^\pi$ and $\mathcal{T}^*$ are γ-contractions in $\|\cdot\|_\infty$ (Theorem 4.3), so the theorem gives existence and uniqueness of V^π and V^* and the convergence of policy evaluation and value iteration in one stroke. Theorem 4.4 states the same result in the notation of Chapter 4.

Example A.2 Contraction on the Trail

Value iteration on the Trail with $\gamma = 0.9$, starting from $V_0 = 0$, updates $V(\text{Goal}) = 4$, $V(\text{Middle}) = \max\{2, 0.9\, V(\text{Goal})\}$ and $V(\text{Start}) = \max\{1, 0.9\, V(\text{Middle})\}$ from the previous iterate. The iterates are $(V(\text{Start}), V(\text{Middle}), V(\text{Goal})) = (1, 2, 4)$, then $(1.8, 3.6, 4)$, then $(3.24, 3.6, 4)$, which is V^*. Measured in the max norm, the distances to V^* are 4, 2.24, 1.44 and 0 for $V_0, \dots, V_3$: each is at most 0.9 times the one before, as the theorem promises (here the decrease is even faster, and the iteration ends after three steps, because the Trail has no loops).

♦

A.5 Optimization

A.5.1 Gradients and gradient descent

For a differentiable $f : \mathbb{R}^d \to \mathbb{R}$, the *gradient* $\nabla f(x)$ is the vector of partial derivatives; it points in the direction of steepest increase, and the first-order Taylor expansion $f(x+\delta) \approx f(x) + \nabla f(x)^\top \delta$

is accurate for small δ. The *chain rule* for compositions, $\nabla_x f(g(x)) = J_g(x)^\top \nabla f(g(x))$ with J_g the Jacobian of g, is what backpropagation computes (Appendix C).

Gradient descent minimizes f by repeating $x_{k+1} = x_k - \eta \nabla f(x_k)$ (gradient *ascent*, with a plus sign, maximizes). If ∇f is L-Lipschitz, $\|\nabla f(x) - \nabla f(y)\|_2 \le L\|x - y\|_2$, then every step with $\eta \le 1/L$ decreases f:

$$f(x_{k+1}) \le f(x_k) - \frac{\eta}{2}\|\nabla f(x_k)\|_2^2.$$

A function is *convex* if $f(\lambda x + (1-\lambda)y) \le \lambda f(x) + (1-\lambda) f(y)$ for $\lambda \in [0,1]$; for differentiable f this is equivalent to $f(y) \ge f(x) + \nabla f(x)^\top (y - x)$, so every stationary point of a convex function is a global minimum. For convex L-smooth f, gradient descent with $\eta = 1/L$ reaches accuracy ϵ in $O(1/\epsilon)$ steps; if f is also μ-strongly convex the error shrinks geometrically, by a factor of about $1 - \mu/L$ per step (Boyd and Vandenberghe 2004). The objectives of reinforcement learning are usually *not* convex; Chapter 14 explains why policy gradient methods nevertheless find global optima in tabular problems.

Stochastic gradient descent (SGD) replaces $\nabla f(x_k)$ by an unbiased estimate g_k with $\mathbb{E}[g_k \mid x_k] = \nabla f(x_k)$, such as the gradient on a random mini-batch. With step sizes satisfying the Robbins–Monro conditions $\sum_k \eta_k = \infty$ and $\sum_k \eta_k^2 < \infty$ it converges under mild conditions (Appendix B). Almost every learning rule in this book has the form *new estimate* = *old estimate* + *step size* × *error*, which is SGD on some objective, or a close relative.

A.5.2 Constraints, Lagrange multipliers and duality

To maximize $f(x)$ subject to constraints $g_i(x) \le 0$ and $h_j(x) = 0$, form the *Lagrangian*

$$\mathcal{L}(x, \lambda, \nu) = f(x) - \sum_i \lambda_i g_i(x) - \sum_j \nu_j h_j(x), \qquad \lambda_i \ge 0.$$

For any feasible x and any $\lambda \ge 0$, $\mathcal{L}(x, \lambda, \nu) \ge f(x)$, so the *dual function* $q(\lambda, \nu) = \max_x \mathcal{L}(x, \lambda, \nu)$ upper-bounds the best feasible value (*weak duality*). For convex problems that have a strictly feasible point (Slater's condition), the bound is tight (*strong duality*), and optimal solutions satisfy the *KKT conditions*: stationarity of $\mathcal{L}$ in x, feasibility, $\lambda \ge 0$, and *complementary slackness* $\lambda_i g_i(x) = 0$ (a multiplier can be positive only on an active constraint) (Boyd and Vandenberghe 2004). Constrained MDPs (Chapter 23) are solved this way, alternating between improving the policy for fixed multipliers and adjusting the multipliers.

Example A.3 The softmax maximizes entropy plus expected value

Among all distributions p over K actions, which one maximizes $\sum_a p_a q_a + \tau \mathcal{H}(p)$, where q_a are action values, $\tau > 0$ is a temperature and $\mathcal{H}(p) = -\sum_a p_a \log p_a$ is the entropy? With a multiplier ν for the constraint $\sum_a p_a = 1$, stationarity of the Lagrangian in p_a gives

$$q_a - \tau(\log p_a + 1) - \nu = 0 \quad \Longrightarrow \quad p_a \propto \exp(q_a/\tau),$$

the softmax (Boltzmann) distribution; the multiplier only fixes the normalization. The optimal value is $\tau \log \sum_a \exp(q_a/\tau)$, the "soft maximum", which tends to $\max_a q_a$ as $\tau \to 0$. This calcula-

tion reappears in soft actor-critic ((13.6)) and in the KL-regularized objective of language-model training ((22.2)).

♦

A closely related calculation, with a KL divergence to a reference distribution π_0 in place of the entropy, gives

$$\arg\max_{p}\Big\{\sum_a p_a q_a - \tau D_{\mathrm{KL}}(p \,\|\, \pi_0)\Big\} = \frac{\pi_0(a)\exp(q_a/\tau)}{\sum_b \pi_0(b)\exp(q_b/\tau)}, \tag{A.2}$$

a *multiplicative-weights* update: the reference probabilities are reweighted by exponentiated values. Natural policy gradient and policy mirror descent (Section 14.4) take exactly this form.

A.5.3 Linear programming

A *linear program* optimizes a linear function subject to linear constraints, for example $\min_x c^\top x$ subject to $Ax \geq b$. Every linear program has a *dual*, here $\max_{y \geq 0} b^\top y$ subject to $A^\top y = c$, with the same optimal value when either is feasible and bounded. Linear programs are solved efficiently in practice by the simplex method and in polynomial time by interior-point methods. In this book they solve MDPs, with occupancy measures as the dual variables (Section 3.10.3), and zero-sum matrix games (Chapter 21).

A.6 Information theory

The *entropy* of a discrete distribution p measures its uncertainty:

$$\mathcal{H}(p) = -\sum_x p(x)\log p(x),$$

with $0 \leq \mathcal{H}(p) \leq \log K$ for K outcomes; it is 0 for a deterministic distribution and $\log K$ for the uniform one. For a Gaussian $\mathcal{N}(\mu, \sigma^2)$, the (differential) entropy is $\frac{1}{2}\log(2\pi e\sigma^2)$. Entropy bonuses keep policies exploratory (Chapter 11 and Chapter 13).

The *Kullback–Leibler divergence* from q to p is

$$D_{\mathrm{KL}}(p \,\|\, q) = \sum_x p(x)\log\frac{p(x)}{q(x)} = \mathbb{E}_{x\sim p}\Big[\log\frac{p(x)}{q(x)}\Big].$$

It is the expected extra cost, in nats, of coding samples of p with a code designed for q. Its key properties:

- $D_{\mathrm{KL}}(p \,\|\, q) \geq 0$, with equality only if $p = q$ (*Gibbs' inequality*; proof below).
- It is not symmetric: $D_{\mathrm{KL}}(p \,\|\, q) \neq D_{\mathrm{KL}}(q \,\|\, p)$ in general. Minimizing $D_{\mathrm{KL}}(p \,\|\, q)$ over q ("forward" KL) makes q cover every region where p has mass; minimizing $D_{\mathrm{KL}}(q \,\|\, p)$ ("reverse" KL) tends to make q concentrate on a single mode of p. This difference matters in imitation learning and in fine-tuning language models.
- It is infinite if $q(x) = 0$ where $p(x) > 0$.

- For Bernoulli distributions, $D_{\mathrm{KL}}(\mathrm{Ber}(a) \,\|\, \mathrm{Ber}(b)) = a \log \frac{a}{b} + (1-a) \log \frac{1-a}{1-b}$; for Gaussians with equal variance, $D_{\mathrm{KL}}(\mathcal{N}(\mu_1, \sigma^2) \,\|\, \mathcal{N}(\mu_2, \sigma^2)) = \frac{(\mu_1 - \mu_2)^2}{2\sigma^2}$.

The *cross-entropy* $-\sum_x p(x) \log q(x) = \mathcal{H}(p) + D_{\mathrm{KL}}(p \,\|\, q)$ is the usual classification loss; minimizing it over q is the same as minimizing the KL divergence from the data distribution.

The *total variation distance* $D_{\mathrm{TV}}(p, q) = \frac{1}{2} \sum_x |p(x) - q(x)| = \max_A |p(A) - q(A)|$ is the largest difference in probability that p and q assign to any event. *Pinsker's inequality* links the two measures:

$$D_{\mathrm{TV}}(p, q) \le \sqrt{\tfrac{1}{2} D_{\mathrm{KL}}(p \,\|\, q)}. \tag{A.3}$$

Trust-region methods use it to turn a bound on the KL divergence between successive policies into a bound on how much their behavior can differ (Theorem 12.3).

The *mutual information* between random variables X and Y is $I(X; Y) = D_{\mathrm{KL}}\big(p(x, y) \,\|\, p(x)\, p(y)\big)$, the amount of information one carries about the other; it is zero exactly when they are independent. Skill-discovery methods maximize it between skills and states (Chapter 20).

A.6.1 Jensen's inequality

If φ is convex, then $\varphi(\mathbb{E}[X]) \le \mathbb{E}[\varphi(X)]$; for concave φ the inequality reverses. Two uses in this book:

- *Gibbs' inequality.* Since log is concave, $-D_{\mathrm{KL}}(p \,\|\, q) = \mathbb{E}_p[\log(q/p)] \le \log \mathbb{E}_p[q/p] = \log \sum_x q(x) \le \log 1 = 0$.
- *Maximization bias.* Since max is convex, $\mathbb{E}[\max_a \hat{Q}(a)] \ge \max_a \mathbb{E}[\hat{Q}(a)]$: the maximum of unbiased noisy estimates is biased upward. This is why Q-learning overestimates and why Double Q-learning helps (Section 6.7).

A.7 Exercises

Exercise A.1 ● Variance of a Bernoulli average

A bandit arm pays 1 with probability p and 0 otherwise. Compute the mean and variance of the average of n independent pulls, and the number of pulls needed for a standard error of 0.01 when $p = 0.3$.

Exercise A.2 ● The tower rule on the Trail

For the uniform policy of the Trail with $\gamma = 0.9$, compute $\mathbb{E}[G_0 \mid a_0 = \mathsf{go}]$ and $\mathbb{E}[G_0 \mid a_0 = \mathsf{exit}]$, and check that averaging them with the probabilities of the two actions gives $V^\pi(\mathsf{Start}) = 1.76$.

Exercise A.3 ●● The score has mean zero

For a softmax distribution $\pi_\theta(a) = e^{\theta_a} / \sum_b e^{\theta_b}$, compute $\nabla_\theta \log \pi_\theta(a)$ and verify directly that $\sum_a \pi_\theta(a) \nabla_\theta \log \pi_\theta(a) = 0$.

Exercise A.4 ●● A stationary distribution

A two-state chain moves from state 1 to state 2 with probability α and from state 2 to state 1 with probability β (otherwise it stays). Find its stationary distribution, and its discounted visitation distribution from state 1 for $\gamma = 0.9$, $\alpha = 0.5$ and $\beta = 0.1$.

Exercise A.5 ●● A projection

Let $\Phi = (1, 1, 1)^\top$ (a single constant feature on three states) and $d = (0.5, 0.25, 0.25)$. Compute Πv for $v = (1, 2, 4)$ and verify the Pythagorean identity.

Exercise A.6 ●● The error bound is tight

Find a γ-contraction on $\mathbb{R}$ and a starting point for which the bound $\|x_k - x^*\| \le \frac{\gamma}{1-\gamma}\|x_k - x_{k-1}\|$ of Theorem A.2 holds with equality for every k.

Exercise A.7 ●● Forward and reverse KL

Let $p = \mathrm{Ber}(0.5)$ and $q = \mathrm{Ber}(0.9)$. Compute $D_{\mathrm{KL}}(p \,\|\, q)$, $D_{\mathrm{KL}}(q \,\|\, p)$ and $D_{\mathrm{TV}}(p, q)$, and check Pinsker's inequality in both directions.

Exercise A.8 ●●● The KL-regularized maximizer

Derive (A.2) with a Lagrange multiplier, and show that the optimal value equals

$$\tau \log \sum_a \pi_0(a)\, e^{q_a/\tau}.$$

What happens as $\tau \to 0$ and as $\tau \to \infty$?

APPENDIX B

Stochastic Approximation and Concentration

Almost every learning rule in this book has the form *new estimate* = *old estimate* + *step size* × *(noisy target* − *old estimate)*. The incremental average of a bandit ((2.2)), TD(0), Q-learning, stochastic gradient descent and the multiplier update of a Lagrangian method (Algorithm 23.2) are all of this kind. Two mathematical tools answer the two questions such rules raise. *Stochastic approximation* says when the iterates converge, and to what: Section B.2 gives the Robbins–Monro conditions on the step sizes, Section B.3 the method that reduces a noisy iteration to an ordinary differential equation, Section B.4 the asynchronous version behind tabular Q-learning, and Section B.5 the case of two coupled iterations. *Concentration inequalities* (Section B.6) say how far an average of n noisy samples can be from its mean, with a bound that holds for every n, not only in the limit; they are the source of exploration bonuses and of every finite-sample guarantee in the book. Section B.1 first introduces the language both tools share. The notation of Appendix A is used throughout; in particular, conditional expectations and the tower rule are reviewed in Section A.1.

B.1 Martingales

A learning algorithm produces a sequence of random quantities, and at each time it may use only what it has seen so far. A *filtration* $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \cdots$ formalizes "what has been seen": $\mathcal{F}_t$ is the information available at time t (in practice, the history $s_0, a_0, r_0, \dots, s_t$ and the algorithm's own random choices). A random variable is *$\mathcal{F}_t$-measurable* if its value is determined by that information.

A sequence $Z_0, Z_1, \dots$ with $\mathbb{E}|Z_t| < \infty$, each Z_t being $\mathcal{F}_t$-measurable, is a *martingale* if

$$\mathbb{E}[Z_{t+1} \mid \mathcal{F}_t] = Z_t \quad \text{for all } t.$$

In words: given everything known now, the next value is expected to equal the current one. The increments $M_{t+1} = Z_{t+1} - Z_t$ then satisfy $\mathbb{E}[M_{t+1} \mid \mathcal{F}_t] = 0$; a sequence with this property is a *martingale difference sequence*. The noise in a stochastic approximation algorithm is almost always of this kind, and this one property replaces independence in most arguments.

Martingale differences are *uncorrelated*: for $s < t$, the tower rule gives $\mathbb{E}[M_{s+1}M_{t+1}] = \mathbb{E}\big[M_{s+1}\,\mathbb{E}[M_{t+1} \mid \mathcal{F}_t]\big] = 0$, because M_{s+1} is known at time t. Consequently the variance of a

sum of martingale differences is the sum of their variances,

$$\mathbb{E}\Big[\Big(\sum_{t=0}^{n-1} M_{t+1}\Big)^2\Big] = \sum_{t=0}^{n-1} \mathbb{E}[M_{t+1}^2],$$

exactly as for independent variables. A fundamental theorem says that a martingale whose second moments stay bounded, $\sup_t \mathbb{E}[Z_t^2] < \infty$, converges with probability one (Kushner and Yin 2003).

Example B.1 The TD errors of the true value function form a martingale difference sequence

Fix a policy π in a finite MDP and let $\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$ be the TD error of the *true* value function, with $V^\pi(\text{terminal}) = 0$. Given the history up to s_t, the Bellman equation says $\mathbb{E}[r_t + \gamma V^\pi(s_{t+1}) \mid \mathcal{F}_t] = V^\pi(s_t)$, so $\mathbb{E}[\delta_t \mid \mathcal{F}_t] = 0$. Summing the discounted TD errors of an episode that ends at time T telescopes:

$$\sum_{t=0}^{T-1} \gamma^t \delta_t = \sum_{t=0}^{T-1} \gamma^t r_t - V^\pi(s_0) = G_0 - V^\pi(s_0).$$

So the error of a Monte Carlo return as an estimate of the value is a sum of martingale differences, one per step. On the Trail with the uniform policy and $\gamma = 0.9$, the episode "go, exit" has $\delta_0 = 0 + 0.9 \times 2.8 - 1.76 = 0.76$ and $\delta_1 = 2 - 2.8 = -0.8$, and indeed $\delta_0 + 0.9\,\delta_1 = 0.04 = 1.8 - 1.76$. Martingale tools such as Theorem B.10 apply directly to such sums.

♦

B.2 The Robbins–Monro algorithm

B.2.1 The problem and the conditions

Robbins and Monro (1951) asked how to find a root x^* of a function h, $h(x^*) = 0$, when $h(x)$ cannot be computed, only measured with noise. Their answer is the iteration

$$x_{t+1} = x_t + \eta_t\big(h(x_t) + M_{t+1}\big), \tag{B.1}$$

where $h(x_t) + M_{t+1}$ is the noisy measurement available at time t and (M_{t+1}) is a martingale difference sequence. Three examples show the range.

- *Estimating a mean.* To find $\mu = \mathbb{E}[\xi]$, take $h(x) = \mu - x$ and measurement $\xi_t - x_t$, so $M_{t+1} = \xi_t - \mu$. The iteration is $x_{t+1} = x_t + \eta_t(\xi_t - x_t)$, the incremental update of a bandit. With $\eta_t = 1/(t+1)$ and $x_0 = 0$, x_t is exactly the average of $\xi_0, \dots, \xi_{t-1}$ (Exercise B.1).
- *Stochastic gradient descent.* To minimize f, take $h = -\nabla f$ and an unbiased gradient estimate; the root is a stationary point of f.
- *TD(0).* With the state s_t to be updated chosen from the past, $h(V) = \mathcal{T}^\pi V - V$ in the updated component, and the noise is the difference between the sampled target $r_t + \gamma V(s_{t+1})$ and its expectation. The root is V^π.

The step sizes must balance two requirements, which are the *Robbins–Monro conditions*:

$$\sum_{t=0}^{\infty} \eta_t = \infty, \qquad \sum_{t=0}^{\infty} \eta_t^2 < \infty. \tag{B.2}$$

The first ensures that the iterates can travel any distance: if $\sum_t \eta_t$ were finite, an unlucky start could never be corrected. The second ensures that the noise is eventually averaged away: the accumulated noise $\sum_t \eta_t M_{t+1}$ has variance $\sum_t \eta_t^2 \mathbb{E}[M_{t+1}^2]$, which must be finite for the noise to settle. The schedules $\eta_t = (t+1)^{-\omega}$ satisfy both conditions exactly when $\frac{1}{2} < \omega \le 1$.

A *constant* step size η violates the second condition, and the iterates never settle: estimating a mean with $x_{t+1} = x_t + \eta(\xi_t - x_t)$, where $\mathrm{Var}(\xi) = \sigma^2$, gives estimates whose variance tends to $\eta\sigma^2/(2-\eta)$ (Exercise B.2). Constant step sizes are nevertheless common in practice, because they track a target that moves, which in reinforcement learning it usually does. The left panel of Figure B.1 shows the three behaviors: with $\eta_t = 1/(t+1)$ the error falls like $1/\sqrt{t}$, the best possible rate; with $(t+1)^{-0.6}$ it falls more slowly; with $\eta = 0.1$ it stops falling at the predicted level $\sqrt{0.1/1.9} = 0.23$.

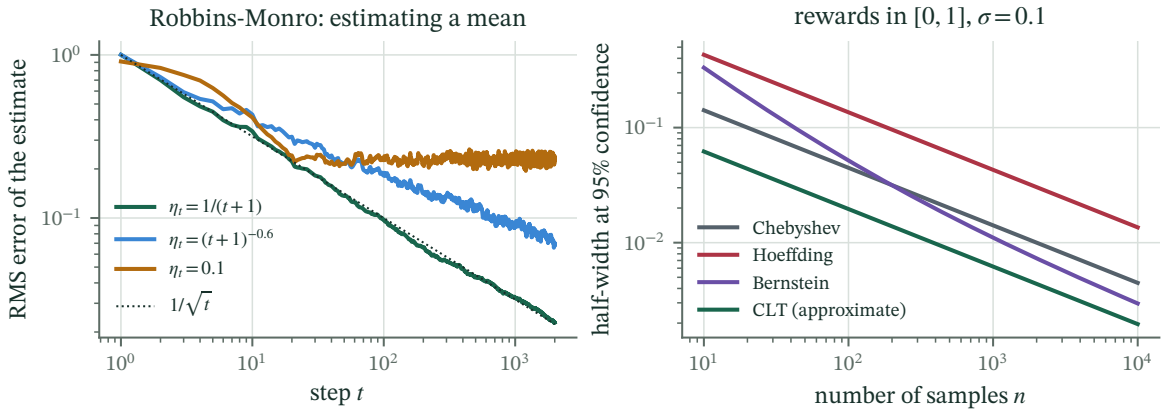


Figure B.1. Left: root-mean-square error of three step-size schedules estimating the mean of $\mathcal{N}(1,1)$ samples (200 runs each). The schedule $1/(t+1)$ gives the sample average and its $1/\sqrt{t}$ error; $(t+1)^{-0.6}$ also converges, more slowly; the constant step 0.1 levels off at $\sqrt{0.1/1.9} \approx 0.23$. Right: half-widths of 95% confidence intervals for the mean of n rewards in $[0,1]$ with standard deviation 0.1, from three inequalities and the central limit theorem. Bernstein's inequality exploits the small variance; Hoeffding's does not; the central limit theorem gives the narrowest, but only approximate, interval.

Averaging helps too. *Polyak–Ruppert averaging* runs the iteration with slowly decreasing step sizes, such as $(t+1)^{-0.6}$, and reports the average $\bar{x}_t = \frac{1}{t}\sum_{k<t} x_k$ of the iterates; under suitable conditions the average achieves the best possible asymptotic variance, as if the optimal step sizes had been known (Polyak and Juditsky 1992).

B.2.2 A convergence theorem

We prove convergence in a simple but useful case: h points toward the root everywhere, and the noise has bounded conditional variance.

Theorem B.1 Robbins–Monro convergence

Let $h : \mathbb{R}^d \to \mathbb{R}^d$ have a root x^* and satisfy, for some constants $c > 0$ and L and all x,

$$\langle x - x^*, h(x)\rangle \le -c\,\|x - x^*\|_2^2, \qquad \|h(x)\|_2 \le L\,\|x - x^*\|_2.$$

Let the noise satisfy $\mathbb{E}[M_{t+1} \mid \mathcal{F}_t] = 0$ and $\mathbb{E}\big[\|M_{t+1}\|_2^2 \mid \mathcal{F}_t\big] \le \sigma^2$, and let the deterministic step sizes $\eta_t \ge 0$ satisfy (B.2). Then, from every starting point x_0, the iterates of (B.1) satisfy $\mathbb{E}\|x_t - x^*\|_2^2 \to 0$ and $x_t \to x^*$ with probability one.

The first condition is a *strong monotonicity* condition. For the mean example, $h(x) = \mu - x$ satisfies it with $c = L = 1$; for gradient descent it holds when f is strongly convex with a Lipschitz gradient. The proof uses two lemmas, the first about deterministic sequences and the second, due to Robbins and Siegmund (1971), about nonnegative random sequences that almost decrease.

Lemma B.2 A deterministic recursion

Let $u_t \ge 0$ satisfy $u_{t+1} \le (1 - a_t)\,u_t + b_t$ with $a_t \in [0,1]$, $\sum_t a_t = \infty$, $b_t \ge 0$ and $\sum_t b_t < \infty$. Then $u_t \to 0$.

Proof. Unrolling the recursion from a time s and using $\prod(1 - a_k) \le 1$,

$$u_T \le \Big[\prod_{k=s}^{T-1}(1 - a_k)\Big]u_s + \sum_{k=s}^{T-1} b_k \le \exp\Big(-\sum_{k=s}^{T-1} a_k\Big)u_s + \sum_{k \ge s} b_k,$$

where $1 - a \le e^{-a}$ was used. Given $\epsilon > 0$, choose s with $\sum_{k \ge s} b_k < \epsilon$; as $T \to \infty$ the first term tends to zero because $\sum_k a_k = \infty$. So $\limsup_T u_T \le \epsilon$ for every $\epsilon > 0$. ∎

Lemma B.3 Robbins–Siegmund

Let V_t, a_t, b_t and u_t be nonnegative $\mathcal{F}_t$-measurable random variables with

$$\mathbb{E}[V_{t+1} \mid \mathcal{F}_t] \le (1 + a_t)\,V_t + b_t - u_t$$

and $\sum_t a_t < \infty$, $\sum_t b_t < \infty$ with probability one. Then, with probability one, V_t converges to a finite limit and $\sum_t u_t < \infty$.

The lemma is a supermartingale convergence theorem with perturbations; its proof uses the martingale convergence theorem (Robbins and Siegmund 1971).

Proof of Theorem B.1. Let $e_t = x_t - x^*$. Then $e_{t+1} = e_t + \eta_t(h(x_t) + M_{t+1})$ and

$$\|e_{t+1}\|^2 = \|e_t\|^2 + 2\eta_t\langle e_t, h(x_t)\rangle + 2\eta_t\langle e_t, M_{t+1}\rangle + \eta_t^2\|h(x_t) + M_{t+1}\|^2.$$

Take the conditional expectation given $\mathcal{F}_t$. The third term vanishes because $\mathbb{E}[M_{t+1} \mid \mathcal{F}_t] = 0$ and e_t is known at time t. In the last term the cross product also vanishes, so $\mathbb{E}[\|h(x_t) + M_{t+1}\|^2 \mid \mathcal{F}_t] = \|h(x_t)\|^2 + \mathbb{E}[\|M_{t+1}\|^2 \mid \mathcal{F}_t] \le L^2\|e_t\|^2 + \sigma^2$. With the monotonicity condition,

$$\mathbb{E}\big[\|e_{t+1}\|^2 \mid \mathcal{F}_t\big] \le \big(1 - 2c\eta_t + L^2\eta_t^2\big)\|e_t\|^2 + \sigma^2\eta_t^2. \tag{B.3}$$

Mean square. Take expectations and write $u_t = \mathbb{E}\|e_t\|^2$. Since $\eta_t \to 0$ (its squares are summable), there is a t_0 after which $L^2\eta_t^2 \le c\eta_t$ and $c\eta_t \le 1$; then $u_{t+1} \le (1 - c\eta_t)u_t + \sigma^2\eta_t^2$. Lemma B.2 with $a_t = c\eta_t$ and $b_t = \sigma^2\eta_t^2$ gives $u_t \to 0$.

Almost surely. Inequality (B.3) has the form of Lemma B.3 with $V_t = \|e_t\|^2$, $a_t = L^2\eta_t^2$, $b_t = \sigma^2\eta_t^2$ and $u_t = 2c\eta_t\|e_t\|^2$. So $\|e_t\|^2$ converges with probability one to some limit V_∞, and $\sum_t \eta_t\|e_t\|^2 < \infty$. If $V_\infty > 0$, then $\|e_t\|^2 \ge V_\infty/2$ for all large t, and $\sum_t \eta_t\|e_t\|^2 = \infty$ because $\sum_t \eta_t = \infty$, a contradiction. Hence $V_\infty = 0$ with probability one, that is, $x_t \to x^*$. ■

The same proof works when the step sizes are random but $\mathcal{F}_t$-measurable and satisfy (B.2) with probability one, and when the bound on the noise grows like $\sigma^2(1 + \|x_t\|^2)$; both extensions are needed for reinforcement learning, where the step size of a state depends on how often it has been visited.

B.3 The ODE method

Strong monotonicity is too much to ask in general. A more powerful view treats the iteration as a noisy discretization of a differential equation. Define the *algorithmic time* $\tau_t = \sum_{k<t} \eta_k$. Over an interval of algorithmic time of length Δ, the iterates take many small steps; the drift terms add up to about $\Delta\, h(x)$, while the noise terms, being uncorrelated with small weights, add up to something negligible. So the iterates follow the trajectories of the *ordinary differential equation* (ODE)

$$\dot{x}(\tau) = h\big(x(\tau)\big), \tag{B.4}$$

and if every trajectory of the ODE converges to x^*, the iterates should too. A point x^* is a *globally asymptotically stable equilibrium* of (B.4) if $h(x^*) = 0$, every trajectory converges to x^*, and trajectories that start near x^* stay near it. The following theorem makes the heuristic precise; its last condition rules out iterates that escape to infinity, which the ODE heuristic alone cannot.

Theorem B.4 The ODE method (Borkar and Meyn)

Consider the iteration (B.1) in $\mathbb{R}^d$ and assume:

1. h is Lipschitz continuous;
2. the step sizes are deterministic, positive and satisfy (B.2);
3. (M_{t+1}) is a martingale difference sequence with $\mathbb{E}\big[\|M_{t+1}\|^2 \mid \mathcal{F}_t\big] \le K(1 + \|x_t\|^2)$ for a constant K;
4. the limit $h_\infty(x) = \lim_{c\to\infty} h(cx)/c$ exists for every x, uniformly on compact sets, and the origin is a globally asymptotically stable equilibrium of $\dot{x} = h_\infty(x)$;
5. the ODE (B.4) has a unique globally asymptotically stable equilibrium x^*.

Then $\sup_t \|x_t\| < \infty$ and $x_t \to x^*$ with probability one.

The theorem combines a convergence result for bounded iterates with the stability criterion of Borkar and Meyn (2000) (condition 4); Borkar (2008) gives a full treatment, including the general

case in which the ODE has several limit sets and the iterates converge to one of them. Condition 4 is easy to check for linear iterations, which is the main use in this book.

Example B.2 Linear TD learning

TD(0) with linear features $V_w(s) = w^\top \phi(s)$, with states sampled independently from a distribution d, is the iteration $w_{t+1} = w_t + \eta_t\big(r_t + \gamma\phi(s'_t)^\top w_t - \phi(s_t)^\top w_t\big)\phi(s_t)$. Its mean update is $h(w) = b - Aw$ with $A = \mathbb{E}[\phi(s)(\phi(s) - \gamma\phi(s'))^\top]$ and $b = \mathbb{E}[r\,\phi(s)]$; then $h_\infty(w) = -Aw$. If every eigenvalue of A has a positive real part, both $\dot{w} = b - Aw$ and $\dot{w} = -Aw$ have globally asymptotically stable equilibria ($A^{-1}b$ and 0), and the theorem gives convergence to the TD fixed point $A^{-1}b$. When d is the stationary distribution of the policy being evaluated and the features are linearly independent, $x^\top Ax > 0$ for every $x \neq 0$, which implies the eigenvalue condition; this is the core of the convergence proof of on-policy linear TD (Theorem 9.4). With off-policy sampling, A can have an eigenvalue with negative real part, the ODE diverges, and so does TD (Section 9.8; Exercise B.6).

♦

In reinforcement learning the samples usually come from a single trajectory of a Markov chain rather than independently, so the noise is not a martingale difference sequence. The ODE method extends to such *Markov noise* under mixing conditions; the analysis of linear TD by Tsitsiklis and Van Roy (1997) is of this kind, and Benveniste et al. (1990) and Borkar (2008) treat the general theory.

B.4 Asynchronous stochastic approximation

Tabular Q-learning updates one entry of its table at a time, the one for the state and action just visited, and leaves the rest unchanged. Its target is a sample of the Bellman optimality operator, which is a contraction in the max norm rather than a strongly monotone function. The following theorem, the result used to prove that tabular Q-learning converges (Theorem 6.1), is made for exactly this situation.

Theorem B.5 Asynchronous stochastic approximation for contractions

Let $F : \mathbb{R}^n \to \mathbb{R}^n$ and $x^* \in \mathbb{R}^n$ satisfy $\|F(x) - x^*\|_\infty \leq \gamma \|x - x^*\|_\infty$ for all x, with $\gamma \in [0, 1)$; every γ-contraction in the max norm qualifies, with x^* its fixed point. Consider the iteration, for each component $i = 1, \dots, n$,

$$x_{t+1}(i) = x_t(i) + \eta_t(i)\big(F(x_t)(i) - x_t(i) + w_t(i)\big),$$

where the step sizes $\eta_t(i) \in [0, 1]$ are $\mathcal{F}_t$-measurable ($\eta_t(i) = 0$ when component i is not updated at time t) and $w_t(i)$ is $\mathcal{F}_{t+1}$-measurable. Assume that, with probability one, for every i:

1. $\sum_t \eta_t(i) = \infty$ and $\sum_t \eta_t(i)^2 < \infty$;
2. $\mathbb{E}[w_t(i) \mid \mathcal{F}_t] = 0$ and $\mathbb{E}[w_t(i)^2 \mid \mathcal{F}_t] \leq A + B\,\|x_t\|_\infty^2$ for constants A and B.

Then $x_t \to x^*$ with probability one.

This is the form given by Bertsekas and Tsitsiklis (1996, sec. 4.3); Tsitsiklis (1994) and Jaakkola et al. (1994) proved closely related versions, the former also allowing the update to use outdated components of x.

Proof sketch. We sketch the argument under the extra assumption that the iterates are bounded, $\|x_t\|_\infty \le C$; Tsitsiklis (1994) shows that under the theorem's conditions this holds automatically. Write $x_t - x^*$ as the sum of a part driven by the noise and a part driven by F.

The noise averages out. For a fixed time s and each component, let $W_s(i) = 0$ and $W_{t+1}(i) = (1-\eta_t(i))W_t(i) + \eta_t(i)\, w_t(i)$ for $t \ge s$. This is a Robbins–Monro iteration that estimates the mean 0 from the noisy samples $w_t(i)$, whose conditional variance is bounded by $A + BC^2$; the argument of Theorem B.1 (with random, $\mathcal{F}_t$-measurable steps) shows that $W_t(i) \to 0$ with probability one.

The contraction shrinks a box. Suppose that from some time s on, $\|x_t - x^*\|_\infty \le D$. Then $|F(x_t)(i) - x^*(i)| \le \gamma D$ for $t \ge s$. Let $Y_s(i) = D$ and $Y_{t+1}(i) = (1-\eta_t(i))Y_t(i) + \eta_t(i)\,\gamma D$. Comparing the recursions term by term (the update is an average of the old value and a target, with weights $1-\eta$ and η) shows by induction that

$$-Y_t(i) + W_t(i) \;\le\; x_t(i) - x^*(i) \;\le\; Y_t(i) + W_t(i), \qquad t \ge s.$$

Because $\sum_t \eta_t(i) = \infty$, $Y_t(i) \to \gamma D$; because $W_t(i) \to 0$, eventually $\|x_t - x^*\|_\infty \le \gamma D + \epsilon$ for any $\epsilon > 0$. Choosing $\epsilon = \frac{1-\gamma}{2} D$ gives a new box of radius $D' = \frac{1+\gamma}{2} D < D$, valid from some later time on.

Iterate. Starting from $D_0 = C + \|x^*\|_\infty$ and repeating the argument gives radii $D_k = \left(\frac{1+\gamma}{2}\right)^k D_0 \to 0$, each valid from some (random) time on. So $x_t \to x^*$. ■

Example B.3 Why tabular Q-learning converges

For Q-learning the components are the pairs (s, a), $F = \mathcal{T}^*$ is the Bellman optimality operator for action values, which is a γ-contraction in the max norm (Theorem 4.3), and $x^* = Q^*$. At time t only the visited pair is updated, with $\eta_t(s_t, a_t) = \alpha_n$ if this is the n-th visit to the pair, and the noise is $w_t = r_t + \gamma \max_{a'} Q_t(s_{t+1}, a') - (\mathcal{T}^* Q_t)(s_t, a_t)$. Given the history up to the choice of (s_t, a_t), the next state and reward are drawn from the MDP, so $\mathbb{E}[w_t \mid \mathcal{F}_t] = 0$, and with rewards of bounded variance $\mathbb{E}[w_t^2 \mid \mathcal{F}_t] \le A + B\|Q_t\|_\infty^2$. The step-size condition of the theorem holds if every pair is visited infinitely often and $\sum_n \alpha_n = \infty$, $\sum_n \alpha_n^2 < \infty$ (for example $\alpha_n = 1/n$). The theorem then gives $Q_t \to Q^*$ with probability one.

♦

B.5 Two timescales

Some algorithms run two coupled iterations, one much faster than the other:

$$x_{t+1} = x_t + \alpha_t\big(f(x_t, y_t) + M_{t+1}^{(1)}\big), \qquad y_{t+1} = y_t + \beta_t\big(g(x_t, y_t) + M_{t+1}^{(2)}\big), \qquad \frac{\beta_t}{\alpha_t} \to 0.$$

In actor-critic methods, x is the critic and y the actor (Konda and Tsitsiklis 2000); in gradient-TD methods (Algorithm 9.5), x is an auxiliary weight vector; in Lagrangian methods for constrained MDPs (Algorithm 23.2), x is the policy and y the multiplier. Seen from the slow iterate, the fast one has always converged already; seen from the fast iterate, the slow one stands still.

Theorem B.6 Two-timescale stochastic approximation (Borkar)

Assume that f and g are Lipschitz; that both step-size sequences satisfy (B.2) and $\beta_t/\alpha_t \to 0$; that $M^{(1)}$ and $M^{(2)}$ are martingale difference sequences with conditional second moments bounded by $K(1+\|x_t\|^2+\|y_t\|^2)$; that $\sup_t(\|x_t\|+\|y_t\|) < \infty$ with probability one; that for every fixed y the ODE $\dot{x} = f(x, y)$ has a globally asymptotically stable equilibrium $\lambda(y)$, with λ Lipschitz; and that $\dot{y} = g(\lambda(y), y)$ has a globally asymptotically stable equilibrium y^*. Then $(x_t, y_t) \to (\lambda(y^*), y^*)$ with probability one.

The result is due to Borkar (1997), and Borkar (2008) gives a textbook treatment. The boundedness of the iterates is an assumption of the theorem, not a conclusion, and must be checked separately in each application.

B.6 Concentration inequalities

The law of large numbers says that an average of n samples converges to the mean, and the central limit theorem says that its error is roughly normal with standard deviation $\sigma/\sqrt{n}$ for large n (Section A.1). Neither says anything definite for a particular finite n. A *concentration inequality* does: it bounds the probability that an average deviates from its mean by more than ϵ, for every n. Turned around, it gives a confidence interval that is valid at every sample size, which is what an exploring agent needs when it decides, after a few tries, whether an action might still be the best.

B.6.1 Markov, Chebyshev and Chernoff

Proposition B.7 Markov's and Chebyshev's inequalities

If $X \geq 0$ and $a > 0$, then $\Pr(X \geq a) \leq \mathbb{E}[X]/a$. Consequently, for any X with variance σ^2 and any $\epsilon > 0$, $\Pr(|X - \mathbb{E}X| \geq \epsilon) \leq \sigma^2/\epsilon^2$.

Proof. Since $X \geq 0$, $X \geq a\,\mathbb{1}\{X \geq a\}$; taking expectations gives $\mathbb{E}[X] \geq a\Pr(X \geq a)$. Applying this to the nonnegative variable $(X - \mathbb{E}X)^2$ with $a = \epsilon^2$ gives Chebyshev's inequality. ■

For the average $\bar{X}_n$ of n independent samples with variance σ^2, Chebyshev's inequality gives $\Pr(|\bar{X}_n - \mu| \geq \epsilon) \leq \sigma^2/(n\epsilon^2)$, so a $1-\delta$ confidence interval has half-width $\sigma/\sqrt{n\delta}$. The dependence on δ is poor: halving the failure probability multiplies the width by $\sqrt{2}$. Much better bounds come from applying Markov's inequality to $e^{\lambda S}$ instead of S^2. For any $\lambda > 0$,

$$\Pr(S \geq t) = \Pr\big(e^{\lambda S} \geq e^{\lambda t}\big) \leq e^{-\lambda t}\,\mathbb{E}\big[e^{\lambda S}\big], \tag{B.5}$$

and one then chooses the best λ. This is the *Chernoff method*. When S is a sum of independent terms, $\mathbb{E}[e^{\lambda S}]$ factorizes, and everything reduces to bounding the moment generating function of a single term.

B.6.2 Hoeffding's inequality

Lemma B.8 Hoeffding's lemma

If $a \le X \le b$ and $\mathbb{E}[X] = 0$, then $\mathbb{E}[e^{\lambda X}] \le \exp\big(\lambda^2 (b-a)^2/8\big)$ for every $\lambda \in \mathbb{R}$.

Proof. Since $e^{\lambda x}$ is convex, on $[a,b]$ it lies below the chord: $e^{\lambda x} \le \frac{b-x}{b-a} e^{\lambda a} + \frac{x-a}{b-a} e^{\lambda b}$. Taking expectations and using $\mathbb{E}[X] = 0$,

$$\mathbb{E}[e^{\lambda X}] \le \frac{b}{b-a} e^{\lambda a} - \frac{a}{b-a} e^{\lambda b} = e^{\psi(u)}, \qquad \psi(u) = -\theta u + \log\big(1 - \theta + \theta e^{u}\big),$$

with $u = \lambda(b-a)$ and $\theta = -a/(b-a) \in [0,1]$ (note $a \le 0 \le b$). Direct differentiation gives $\psi(0) = 0$, $\psi'(0) = 0$ and $\psi''(u) = p(1-p)$ with $p = \theta e^u/(1-\theta+\theta e^u) \in [0,1]$, so $\psi'' \le \frac{1}{4}$. By Taylor's theorem, $\psi(u) \le u^2/8 = \lambda^2 (b-a)^2/8$. ∎

Theorem B.9 Hoeffding's inequality

Let $X_1, \dots, X_n$ be independent with $a_i \le X_i \le b_i$, and let $S = \sum_{i=1}^n (X_i - \mathbb{E}X_i)$. Then for every $t > 0$,

$$\Pr(S \ge t) \le \exp\Big(-\frac{2t^2}{\sum_{i=1}^n (b_i - a_i)^2}\Big).$$

In particular, if every X_i lies in $[a,b]$ and has mean μ, the average satisfies $\Pr(\bar X_n - \mu \ge \epsilon) \le e^{-2n\epsilon^2/(b-a)^2}$, and with probability at least $1-\delta$,

$$|\bar X_n - \mu| \le (b-a)\sqrt{\frac{\ln(2/\delta)}{2n}}.$$

Proof. Each centered term $X_i - \mathbb{E}X_i$ lies in an interval of length $b_i - a_i$ and has mean zero. By independence and Lemma B.8, $\mathbb{E}[e^{\lambda S}] = \prod_i \mathbb{E}[e^{\lambda(X_i - \mathbb{E}X_i)}] \le \exp\big(\lambda^2 c/8\big)$ with $c = \sum_i (b_i - a_i)^2$. The Chernoff bound (B.5) gives $\Pr(S \ge t) \le \exp(-\lambda t + \lambda^2 c/8)$, which is smallest at $\lambda = 4t/c$, where it equals $\exp(-2t^2/c)$. For the average, take $t = n\epsilon$ and $c = n(b-a)^2$. The same bound holds for $-S$, so the two-sided probability is at most $2e^{-2n\epsilon^2/(b-a)^2}$; setting this equal to δ and solving for ϵ gives the interval. ∎

For rewards in $[0,1]$, the interval has half-width $\sqrt{\ln(2/\delta)/(2n)}$, about 0.136 with $n = 100$ and $\delta = 0.05$. To be within ± 0.05 of the mean with probability 95% needs $n \ge \ln(40)/(2\times 0.05^2) = 738$ samples, whatever the distribution; Chebyshev's inequality with the largest possible standard deviation, $\sigma = 0.5$, would need 2,000, and the normal approximation 385.

Example B.4 Where the UCB bonus comes from

An arm with rewards in $[0,1]$ has been pulled N times, with sample mean $\hat{\mu}$. By Hoeffding's inequality, $\Pr(\mu \geq \hat{\mu} + \epsilon) \leq e^{-2N\epsilon^2}$. Choosing $\epsilon = \sqrt{2\ln t/N}$ makes this probability t^{-4}. So with probability at least $1 - t^{-4}$, the true mean is below $\hat{\mu} + \sqrt{2\ln t/N}$, the optimistic index of UCB1 (Algorithm 2.2). The probability t^{-4} is small enough that a union bound over all arms and all possible values of $N \leq t$ still gives a failure probability that is summable over t, which is what the regret analysis of Theorem 2.6 needs.

♦

B.6.3 The union bound and simultaneous confidence

The *union bound* $\Pr\big(\bigcup_i A_i\big) \leq \sum_i \Pr(A_i)$ turns statements about one estimate into statements about many. If each of K intervals fails with probability at most δ/K, all K hold together with probability at least $1-\delta$. With Hoeffding's inequality the price is mild: the half-width grows from $\sqrt{\ln(2/\delta)/(2n)}$ to $\sqrt{\ln(2K/\delta)/(2n)}$, only logarithmically in K. This is why exploration algorithms can afford to keep confidence intervals for every state, action and time step (Chapter 16).

B.6.4 Martingale differences: Azuma–Hoeffding

Hoeffding's argument needs only that each term has a bounded range and conditional mean zero given the past, not full independence.

Theorem B.10 Azuma–Hoeffding inequality

Let $D_1, \dots, D_n$ be a martingale difference sequence with respect to $\mathcal{F}_0 \subseteq \cdots \subseteq \mathcal{F}_n$, and suppose that $A_i \leq D_i \leq A_i + c_i$, where A_i is $\mathcal{F}_{i-1}$-measurable and the c_i are constants. Then for every $t > 0$,

$$\Pr\Big(\sum_{i=1}^{n} D_i \geq t\Big) \leq \exp\Big(-\frac{2t^2}{\sum_{i=1}^{n} c_i^2}\Big).$$

In particular, if $|D_i| \leq c_i$, the bound is $\exp\big(-t^2/(2\sum_i c_i^2)\big)$.

Proof. Given $\mathcal{F}_{i-1}$, D_i has mean zero and lies in an interval of length c_i, so Lemma B.8 applies conditionally: $\mathbb{E}[e^{\lambda D_i} \mid \mathcal{F}_{i-1}] \leq e^{\lambda^2 c_i^2/8}$. With $S_k = \sum_{i \leq k} D_i$ and the tower rule,

$$\mathbb{E}[e^{\lambda S_n}] = \mathbb{E}\big[e^{\lambda S_{n-1}}\,\mathbb{E}[e^{\lambda D_n} \mid \mathcal{F}_{n-1}]\big] \leq e^{\lambda^2 c_n^2/8}\,\mathbb{E}[e^{\lambda S_{n-1}}] \leq \cdots \leq \exp\Big(\frac{\lambda^2}{8}\sum_i c_i^2\Big).$$

The Chernoff bound with $\lambda = 4t/\sum_i c_i^2$ finishes the proof, as for Theorem B.9. If $|D_i| \leq c_i$, take $A_i = -c_i$ and an interval of length $2c_i$. ■

The sums of Example B.1 are of this form. Regret analyses of exploration algorithms (Theorem 16.3) apply the inequality to sums of such differences collected over many episodes, where the data are anything but independent, because the policy changes with what was learned.

B.6.5 Bernstein's inequality

Hoeffding's inequality uses only the range of the variables. When the variance is much smaller than the range allows, as for a rare event or a reward that is usually close to its mean, a bound that uses the variance is much sharper.

Theorem B.11 Bernstein's inequality

Let $X_1, \dots, X_n$ be independent with $\mathbb{E}[X_i] = 0$, $|X_i| \le b$ and $\sum_{i=1}^{n} \mathbb{E}[X_i^2] = v$. Then for every $t > 0$,

$$\Pr\Big(\sum_{i=1}^{n} X_i \ge t\Big) \le \exp\Big(-\frac{t^2}{2(v + bt/3)}\Big).$$

Consequently, if the X_i are i.i.d. with mean μ, variance σ^2 and $|X_i - \mu| \le b$, then with probability at least $1 - \delta$,

$$\bar{X}_n - \mu \le \sqrt{\frac{2\sigma^2 \ln(1/\delta)}{n}} + \frac{2b \ln(1/\delta)}{3n}.$$

Proof. *A bound on the moment generating function.* For one term X with $\mathbb{E}X = 0$, $|X| \le b$ and $\mathbb{E}X^2 = \sigma_X^2$, expand the exponential and use $|X|^k \le X^2 b^{k-2}$ for $k \ge 2$:

$$\mathbb{E}[e^{\lambda X}] = 1 + \sum_{k \ge 2} \frac{\lambda^k \mathbb{E}[X^k]}{k!} \le 1 + \frac{\sigma_X^2}{b^2} \sum_{k \ge 2} \frac{(\lambda b)^k}{k!}.$$

For $0 \le \lambda < 3/b$, since $k! \ge 2 \cdot 3^{k-2}$, the sum is at most $\frac{(\lambda b)^2}{2} \sum_{j \ge 0} (\lambda b/3)^j = \frac{(\lambda b)^2}{2(1 - \lambda b/3)}$. With $1 + y \le e^y$, $\mathbb{E}[e^{\lambda X}] \le \exp\big(\frac{\lambda^2 \sigma_X^2}{2(1 - \lambda b/3)}\big)$.

Chernoff. By independence, $\Pr(\sum_i X_i \ge t) \le \exp\big(-\lambda t + \frac{\lambda^2 v}{2(1 - \lambda b/3)}\big)$. Choose $\lambda = t/(v + bt/3)$, which is below $3/b$. Then $1 - \lambda b/3 = v/(v + bt/3)$, the second term equals $t^2/(2(v + bt/3))$, the first is $-t^2/(v + bt/3)$, and their sum is the claimed exponent.

The interval. Apply the bound to $X_i - \mu$ with $v = n\sigma^2$ and $t = n\epsilon$, and set the bound equal to δ: $n\epsilon^2 = 2 \ln(1/\delta)\,(\sigma^2 + b\epsilon/3)$. The positive root of this quadratic in ϵ is $\epsilon = \frac{bL}{3n} + \sqrt{\big(\frac{bL}{3n}\big)^2 + \frac{2\sigma^2 L}{n}}$ with $L = \ln(1/\delta)$, and $\sqrt{x + y} \le \sqrt{x} + \sqrt{y}$ gives the stated bound. ∎

The first term of the interval is the central-limit width with σ in place of the worst case; the second, of order $1/n$, pays for the range and becomes negligible for large n. The right panel of Figure B.1 compares the four bounds for rewards in $[0, 1]$ with standard deviation 0.1 and $\delta = 0.05$ (two-sided, so $\ln(2/\delta)$ in place of $\ln(1/\delta)$). With $n = 100$ the half-widths are 0.136 (Hoeffding), 0.052 (Bernstein), 0.045 (Chebyshev) and 0.020 (the central limit theorem, which is only approximate); with $n = 1{,}000$ they are 0.043, 0.011, 0.014 and 0.006. When the variance is as large as the range allows ($\sigma = 0.5$), Bernstein's half-width at $n = 100$, 0.160, is slightly worse than Hoeffding's. In practice σ is unknown; *empirical Bernstein* bounds replace it by the sample standard deviation at a small extra cost (Maurer and Pontil 2009), and the sharpest exploration algorithms for MDPs are built on them (Chapter 16).

KEY IDEA

Hoeffding: width $\approx (b-a)\sqrt{\ln(2/\delta)/(2n)}$, needs only a bounded range. Bernstein: width $\approx \sigma\sqrt{2\ln(2/\delta)/n}$ plus a $1/n$ term, much better when the variance is small. Azuma: Hoeffding for martingale differences. Union bound: many statements at once for a logarithmic price.

B.7 Further reading

Kushner and Yin (2003) and Borkar (2008) are the standard books on stochastic approximation, the first comprehensive and the second short and based on the ODE method; Benveniste et al. (1990) treats Markov noise. Bertsekas and Tsitsiklis (1996) (chapter 4) covers the results used for reinforcement learning, including asynchronous iterations. For concentration, Boucheron et al. (2013) is the modern reference; Lattimore and Szepesvári (2020) develop exactly the inequalities that bandit and exploration algorithms need.

B.8 Exercises

Exercise B.1 • Step sizes

(a) For which $\omega > 0$ does $\eta_t = (t+1)^{-\omega}$ satisfy both Robbins–Monro conditions? (b) Show that $x_{t+1} = x_t + \frac{1}{t+1}(\xi_t - x_t)$ with $x_0 = 0$ gives $x_t = \frac{1}{t}\sum_{k<t}\xi_k$ for $t \geq 1$. (c) Does the step size $\eta_t = 1/(t+1)$ satisfy the conditions? Does $\eta_t = 1/\sqrt{t+1}$?

Exercise B.2 •• A constant step size

Let $\xi_0, \xi_1, \ldots$ be independent with mean μ and variance σ^2, and $x_{t+1} = x_t + \eta(\xi_t - x_t)$ with a constant $\eta \in (0, 1]$. (a) Show that $\mathbb{E}[x_t] \to \mu$ geometrically fast. (b) Show that $\mathrm{Var}(x_t) \to \eta\sigma^2/(2-\eta)$. (c) For $\sigma = 1$ and $\eta = 0.1$, compare with Figure B.1. (d) Explain the trade-off in choosing η when the mean μ drifts slowly over time.

Exercise B.3 • How many samples?

Rewards lie in $[0, 1]$. (a) How many samples guarantee, by Hoeffding's inequality, that the sample mean is within 0.05 of the true mean with probability at least 0.95? (b) How many does Chebyshev's inequality need if the standard deviation is known to be at most 0.5? (c) How many does the normal approximation suggest, with $\sigma = 0.5$? (d) Why might one still prefer the Hoeffding number in an algorithm that must work for every distribution?

Exercise B.4 •• The UCB bonus and the union bound

A bandit has K arms with rewards in $[0, 1]$. (a) Use Theorem B.9 to show that, for a fixed arm and a fixed number of pulls N, $\Pr\big(\mu \geq \hat{\mu}_N + \sqrt{2\ln t/N}\big) \leq t^{-4}$. (b) Use the union bound to show that,

at time t, all K upper confidence bounds hold for every possible number of pulls $N \in \{1, \dots, t\}$ with probability at least $1 - Kt^{-3}$. (c) Why is it important that $\sum_t Kt^{-3}$ is finite?

Exercise B.5 ●● When the variance is small

An event has probability $p = 0.01$ (say, a crash in a safety test), and we observe $n = 1{,}000$ independent trials. Compute the 95% two-sided half-widths for the estimate of p given by Hoeffding's inequality, by Bernstein's inequality (with $b = 0.99$) and by the normal approximation. For which values of p is Bernstein's bound better than Hoeffding's at this n?

Exercise B.6 ●● Stable and unstable linear TD

Two states have a single feature, $\phi(s_1) = 1$ and $\phi(s_2) = 2$, so $V_w(s) = w\,\phi(s)$, and all rewards are zero. Every transition goes from s_1 to s_2 or from s_2 to s_2 (s_2 is absorbing), and $\gamma \in (0, 1)$. (a) Off-policy: updates are made only on the transition $s_1 \to s_2$. Show that the mean update is $h(w) = -(1 - 2\gamma)w$ and that w diverges for $\gamma > \frac{1}{2}$ unless $w_0 = 0$. (b) Now updates are made on the transitions $s_1 \to s_2$ and $s_2 \to s_2$ with probabilities q and $1 - q$. Find A, and the values of q for which the ODE of Theorem B.4 is stable when $\gamma = 0.9$. (c) Relate your answer to Example B.2.

Exercise B.7 ●●● Averaging the iterates

Simulate the estimation of a mean from $\mathcal{N}(1, 1)$ samples with $\eta_t = (t + 1)^{-0.6}$, and compare the root-mean-square error of the last iterate and of the running average of the iterates with that of the sample mean, over $t \le 10^4$ and 200 runs. Then read Polyak and Juditsky (1992) and explain why averaging recovers the optimal rate, and why averaging is less useful when the target moves, as in reinforcement learning.

APPENDIX C

Neural Networks in a Nutshell

The deep reinforcement-learning methods of Parts III to VII represent value functions, policies and models with neural networks. This appendix collects what those chapters assume: what a multilayer perceptron computes (Section C.1), how backpropagation computes its gradients (Section C.2), how optimizers use them (Section C.3), how networks are initialized, normalized and regularized (Section C.4), the main architectures in brief (Section C.5), the habits that matter specifically in reinforcement learning (Section C.6), and a walk through the course's own small library, `rlcourse.nn`, with a gradient check (Section C.7). Goodfellow et al. (2016) is a thorough textbook treatment of everything here. Gradients and the chain rule are reviewed in Section A.5.

C.1 Multilayer perceptrons

A *multilayer perceptron* (MLP) is a composition of simple layers. Each layer multiplies its input by a weight matrix, adds a bias vector, and applies a fixed nonlinear function, the *activation*, to every component. With an input row vector $a_0 = x \in \mathbb{R}^{n_0}$ and layers $\ell = 1, \dots, m$,

$$z_\ell = a_{\ell-1} W_\ell + b_\ell, \qquad a_\ell = f_\ell(z_\ell), \tag{C.1}$$

where $W_\ell \in \mathbb{R}^{n_{\ell-1} \times n_\ell}$ and $b_\ell \in \mathbb{R}^{n_\ell}$ are the parameters, n_ℓ is the *width* of layer ℓ, and f_ℓ is applied componentwise. The output is a_m. We write vectors as rows and multiply them from the left, as the code does, because then a *batch* of B inputs stacked as the rows of a $B \times n_0$ matrix goes through the same formula unchanged. The intermediate layers are *hidden*; the last one is the *output layer*, and its activation is usually the identity. A network with sizes $[4, 64, 64, 2]$ maps a 4-dimensional observation, through two hidden layers of 64 units, to 2 outputs, for example the action values of CartPole's two actions.

Common activations are listed in Table C.1 and drawn in the left panel of Figure C.1. Without them the network would be a composition of linear maps, hence linear. With them, MLPs are *universal approximators*: one hidden layer with enough units can approximate any continuous function on a compact set as closely as desired (Cybenko 1989; Hornik 1991). The theorem says nothing about how many units are needed or whether gradient descent finds the approximation; in practice, depth and width are chosen by experiment.

Table C.1. Activation functions. Both derivatives used in the library can be computed from the output: $\tanh' = 1 - a^2$ and $\mathrm{ReLU}' = \mathbb{1}\{a > 0\}$.

Activation	$f(z)$	$f'(z)$	Notes
tanh	$\tanh z$	$1 - \tanh^2 z$	bounded, smooth; common in policy networks
ReLU	$\max(z, 0)$	$\mathbb{1}\{z > 0\}$	cheap; units can "die" if $z < 0$ always
sigmoid	$1/(1 + e^{-z})$	$f(z)(1 - f(z))$	outputs in $(0, 1)$; probabilities
softplus	$\log(1 + e^z)$	sigmoid of z	smooth, positive; standard deviations
identity	z	1	output layers for values and logits

Output heads. Reinforcement learning uses a few kinds of outputs. A *value head* is a linear output: $V(s)$ or one $Q(s, a)$ per action. A *categorical policy* outputs one *logit* z_a per action and turns them into probabilities with the softmax, $\pi(a \mid s) = e^{z_a} / \sum_b e^{z_b}$. A *Gaussian policy* for continuous actions outputs a mean $\mu(s)$ and either a state-dependent or a free parameter vector of log standard deviations $\log \sigma$; actions are sampled as $a = \mu + \sigma \odot \epsilon$ with $\epsilon \sim \mathcal{N}(0, I)$, and often squashed by tanh into the action bounds (Algorithm 13.3).

Losses. A network is trained by minimizing a loss averaged over a batch. Value functions use the *squared error* $\frac{1}{2}(\hat{y} - y)^2$ against a target y, or the *Huber loss*, which is quadratic for errors below 1 in size and linear beyond, so that its gradient is the error clipped to $[-1, 1]$ (Exercise C.6); DQN uses it to keep large TD errors from producing huge updates (Chapter 10). Classifiers and reward models use the *cross-entropy*, the negative log-probability of the correct label: for logits z and label y, $-\log \mathrm{softmax}(z)_y$, whose gradient with respect to z is the remarkably simple $\mathrm{softmax}(z) - e_y$ (Exercise C.2). Policy-gradient methods minimize a *surrogate* such as $-\hat{A} \log \pi_\theta(a \mid s)$, whose gradient is the policy-gradient estimate; the surrogate's value itself means little (Chapter 11).

C.2 Backpropagation

Training needs the gradient of the loss with respect to every weight. *Backpropagation* computes all of them at about the cost of two forward passes, by applying the chain rule from the output back to the input and reusing intermediate results (Rumelhart et al. 1986). Suppose the forward pass (C.1) has been run and its activations $a_0, \dots, a_m$ stored, and let $g_m = \partial \mathcal{L} / \partial a_m$ be the gradient of the loss with respect to the output (for the squared error, $g_m = \hat{y} - y$). Going backward, for $\ell = m, m - 1, \dots, 1$:

$$\delta_\ell = g_\ell \odot f_\ell'(z_\ell), \qquad \frac{\partial \mathcal{L}}{\partial W_\ell} = a_{\ell-1}^\top \delta_\ell, \qquad \frac{\partial \mathcal{L}}{\partial b_\ell} = \delta_\ell, \qquad g_{\ell-1} = \delta_\ell W_\ell^\top. \tag{C.2}$$

In words: $\delta_\ell = \partial\mathcal{L}/\partial z_\ell$ is the gradient at the pre-activations of layer ℓ; it gives the weight gradient as an outer product with the layer's input, and it is passed down to the previous layer through the transposed weights. For a batch, the rows of $a_{\ell-1}$ and δ_ℓ are stacked, $a_{\ell-1}^\top \delta_\ell$ sums the outer products over the batch, and the bias gradient is the sum of the rows of δ_ℓ; this is exactly what `MLP.backward` in `rlcourse.nn` computes, and it returns $g_0 = \partial\mathcal{L}/\partial x$ as well.

Derivation. Each equation is one use of the chain rule. Since $a_\ell = f_\ell(z_\ell)$ componentwise, $\partial\mathcal{L}/\partial z_{\ell,j} = (\partial\mathcal{L}/\partial a_{\ell,j})\, f_\ell'(z_{\ell,j})$, which is the first equation. Since $z_{\ell,j} = \sum_i a_{\ell-1,i} W_{\ell,ij} + b_{\ell,j}$, the derivative of $z_{\ell,j}$ with respect to $W_{\ell,ij}$ is $a_{\ell-1,i}$, with respect to $b_{\ell,j}$ it is 1, and with respect to $a_{\ell-1,i}$ it is $W_{\ell,ij}$; summing over j the paths through which W, b and $a_{\ell-1}$ influence the loss gives the other three.

Example C.1 Backpropagation by hand

Take a network with one input, two tanh hidden units and a linear output, with $W_1 = (0.5, -1)$, $b_1 = (0, 0.5)$, $W_2 = (1, 2)^\top$, $b_2 = 0$, the input $x = 1$, the target $y = 1$ and the loss $\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$.

Forward. $z_1 = (0.5, -0.5)$, $a_1 = (\tanh 0.5, -\tanh 0.5) = (0.4621, -0.4621)$, $\hat{y} = 0.4621 \times 1 - 0.4621 \times 2 = -0.4621$, and $\mathcal{L} = \frac{1}{2}(1.4621)^2 = 1.0689$.

Backward. $g_2 = \hat{y} - y = -1.4621$ and, the output being linear, $\delta_2 = g_2$. So $\partial\mathcal{L}/\partial W_2 = a_1^\top \delta_2 = (-0.6757, 0.6757)^\top$ and $\partial\mathcal{L}/\partial b_2 = -1.4621$. Passing down, $g_1 = \delta_2 W_2^\top = (-1.4621, -2.9242)$; with $1 - a_1^2 = (0.7864, 0.7864)$, $\delta_1 = (-1.1499, -2.2998)$, which is also $\partial\mathcal{L}/\partial W_1$ (the input is 1) and $\partial\mathcal{L}/\partial b_1$. Finally $\partial\mathcal{L}/\partial x = \delta_1 W_1^\top = -0.5749 + 2.2998 = 1.7248$. The test file `labs/tests/test_appC_nn.py` reproduces every number with `rlcourse.nn`.

♦

The gradient with respect to the *input*, which supervised learning rarely needs, is essential for some reinforcement-learning methods: DDPG, TD3 and SAC train the actor by following $\nabla_a Q(s, a)$, the gradient of the critic with respect to its action input (Chapter 13).

C.3 Optimizers

Given gradients, an optimizer updates the parameters θ (all the W_ℓ and b_ℓ together). Write g_k for the minibatch gradient at step k and η for the learning rate.

Stochastic gradient descent (SGD) takes $\theta_{k+1} = \theta_k - \eta\, g_k$ (Section A.5; its convergence theory is stochastic approximation, Appendix B). *Momentum* accumulates a velocity, $m_{k+1} = \beta m_k + g_k$ and $\theta_{k+1} = \theta_k - \eta\, m_{k+1}$ with β around 0.9, which damps oscillations across narrow valleys of the loss and speeds progress along them (Polyak 1964; Sutskever et al. 2013).

Adam (Kingma and Ba 2015), the default optimizer of deep RL, keeps exponential moving averages of the gradient and of its elementwise square, corrects their bias toward zero at the start, and divides:

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k, \quad v_k = \beta_2 v_{k-1} + (1-\beta_2) g_k^2, \quad \theta_k = \theta_{k-1} - \eta\, \frac{m_k/(1-\beta_1^k)}{\sqrt{v_k/(1-\beta_2^k)} + \epsilon},$$

with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and a small ϵ such as 10^{-8} (all operations are componentwise; this ϵ is unrelated to PPO's clip range). Each parameter gets its own effective step size, about η times the ratio of its average gradient to its root-mean-square gradient. A consequence worth remembering: the update hardly changes if the loss is multiplied by a constant, and its size is about η per parameter. At the first step it is exactly $\eta\, g_1/(|g_1| + \epsilon) \approx \eta \operatorname{sign}(g_1)$, whatever the scale of the gradient (Exercise C.4). *AdamW* applies weight decay directly to the parameters rather than through the gradient, which interacts better with the adaptive scaling (Loshchilov and Hutter 2019).

Gradient clipping rescales the whole gradient vector when its norm exceeds a threshold c: $g \leftarrow g \cdot \min\{1, c/\|g\|_2\}$ (Pascanu et al. 2013). It guards against rare huge gradients, from large TD errors or near-zero probabilities, and most PPO implementations clip at a global norm of 0.5 (Section 12.7). Learning rates are often decayed linearly to zero over training, especially in on-policy methods.

The right panel of Figure C.1 trains a $[1, 32, 32, 1]$ tanh network to fit $\sin 3x$ on $[-1, 1]$ from minibatches of 64 points. Adam with $\eta = 0.01$ reaches a mean squared error of 6×10^{-5} after 2,000 steps; SGD with $\eta = 0.1$ reaches 3×10^{-3}, and Adam with the more cautious $\eta = 0.001$ reaches 2×10^{-3}. The learning rate matters as much as the optimizer.

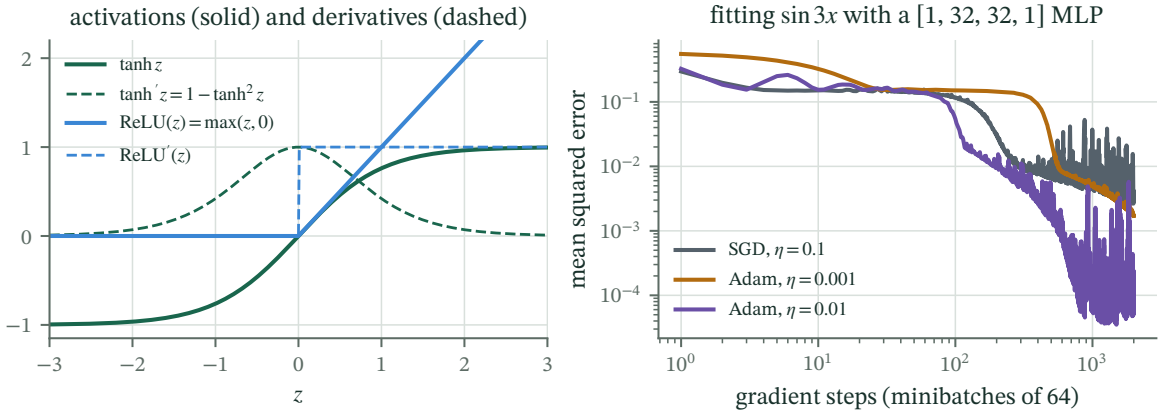


Figure C.1. Left: the tanh and ReLU activations (solid) and their derivatives (dashed). Right: mean squared error while fitting $\sin 3x$ with a $[1, 32, 32, 1]$ tanh network from `rlcourse.nn`, with SGD and with Adam at two learning rates (one run each, the same initialization and minibatches). Adam with a well-chosen learning rate is fastest here; its final error is 30 to 50 times smaller than the others'.

C.4 Initialization, normalization and regularization

Initialization. If the initial weights are too large, activations saturate (tanh units stuck near ± 1, where their derivative vanishes) or explode; if they are too small, signals and gradients shrink layer by layer. Scaled random initializations keep the variance of the activations roughly constant from layer to layer. *Glorot* (or Xavier) initialization draws W_{ij} with variance $2/(n_{\text{in}} + n_{\text{out}})$, a compromise between preserving the variance forward (which needs $1/n_{\text{in}}$) and backward (which needs $1/n_{\text{out}}$) (Glorot and Bengio 2010) (Exercise C.5); `rlcourse.nn` uses it. *He* initialization,

variance $2/n_{\text{in}}$, compensates for ReLU zeroing half of its inputs (He et al. 2015). *Orthogonal* initialization draws a random orthogonal matrix, which preserves norms exactly (Saxe et al. 2014), and is common in PPO implementations. Biases start at zero. In reinforcement learning the output layer of a policy is often scaled down, by a factor such as 0.01, so that the initial policy is close to uniform; `MLP(..., out_scale=0.01)` does this.

Normalization. *Layer normalization* (Ba et al. 2016) rescales each example's hidden activations to zero mean and unit variance across the units of a layer, followed by a learned scale and shift; it does not depend on the other examples in the batch and is widely used in transformers and increasingly in RL networks. *Batch normalization* (Ioffe and Szegedy 2015) normalizes each unit across the examples of a batch. It is awkward in reinforcement learning: the statistics of a batch drawn from a replay buffer differ from those of the states the agent acts in, and the network then behaves differently in training and in acting. Normalizing the *inputs* with running statistics (Section 23.2) is standard and cheap.

Regularization. Weight decay (a penalty on $\|\theta\|_2^2$), dropout (Srivastava et al. 2014) and early stopping fight overfitting in supervised learning. In reinforcement learning, overfitting is usually less of a problem than instability and the gradual *loss of plasticity* of networks trained on a changing stream of targets (Section 24.5); for agents that learn from images, data augmentation such as random shifts is an effective regularizer (Kostrikov et al. 2021).

C.5 Convolutional and recurrent networks, and transformers

Convolutional networks process images with small filters that slide across the input, sharing their weights across positions, so that a feature detected in one place is detected everywhere; pooling or strided convolutions then reduce the resolution (LeCun et al. 1998). The DQN agent for Atari (Mnih et al. 2015) used three convolutional layers (32 filters of size 8×8 with stride 4, 64 of size 4×4 with stride 2, and 64 of size 3×3 with stride 1) followed by a fully connected layer of 512 units, applied to a stack of the last four 84×84 gray-scale frames. Variants of this "Nature CNN", and residual networks for harder visual tasks, remain the standard encoders of pixel-based agents.

Recurrent networks keep a hidden state $h_t = \phi(h_{t-1}, x_t)$ that summarizes the inputs seen so far, which is what an agent needs when the current observation is not a Markov state (Chapter 19). Plain recurrent networks have trouble carrying information over many steps, because gradients propagated back through time shrink or explode; the *LSTM* (Hochreiter and Schmidhuber 1997) and the simpler *GRU* (Cho et al. 2014) use multiplicative gates that let information and gradients pass through many steps. They are trained by *backpropagation through time*: the network is unrolled over a sequence and (C.2) is applied to the unrolled computation. Recurrent agents need care with the hidden state stored in replay buffers (Section 19.5.2).

Transformers (Vaswani et al. 2017) replace recurrence by *attention*. Each element of a sequence emits a query q, a key k and a value v (linear maps of its representation); stacking them as matrices Q, K, V for a sequence, one attention layer computes

$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}}\Big)V,$$

with the softmax taken over each row, so that every output is a weighted average of the values, with weights given by how well its query matches each key; d_k is the dimension of the keys. Several such *heads* run in parallel, and layers alternate attention with small MLPs, residual connections and layer normalization. Transformers are the architecture of language models (Chapter 22) and appear in RL as memory for partially observed tasks and as sequence models of trajectories (Chapter 17).

C.6 Practical tips for reinforcement learning

Networks in reinforcement learning are trained on data they help to generate, against targets that move. A few habits address this.

- Stabilize the targets. Bootstrapped targets computed with the network being trained chase themselves; a *target network*, a lagged copy updated periodically or by Polyak averaging $\bar{w} \leftarrow (1-\tau)\bar{w} + \tau w$, slows the chase (Section 10.2.2). `MLP.copy` and `MLP.soft_update` implement both.
- Keep updates small and bounded. Clip gradients by their global norm, use the Huber loss for TD errors, and prefer Adam with a learning rate around 10^{-4} to 3×10^{-4} as a starting point.
- Start policies near uniform. Scale down the output layer of the policy, so that the agent explores at first and the early gradients are well behaved.
- Normalize inputs and advantages. Use running statistics for observations (Section 23.2); normalize advantages within each batch in policy-gradient methods.
- Separate or share? Sharing a network trunk between actor and critic saves computation, but the critic's loss can dominate the shared features; separate networks are the safer default for small problems.
- Avoid batch normalization unless you know how its statistics will be handled in acting and training; prefer layer normalization.
- Watch for numerical trouble. Compute log-probabilities with `log_softmax` rather than `log(softmax(...))`, bound log standard deviations, and assert that losses and gradients are finite.
- Check gradients once. Compare analytic and numerical gradients on a tiny network (Section C.7) whenever you write a new loss or layer.

C.7 A walk through `rlcourse.nn`

The course library contains a deliberately small neural-network module, `labs/rlcourse/nn.py`, of about 170 lines, so that nothing in the deep RL labs is hidden. Its pieces are the class `MLP` (the network of (C.1), with `forward`, `backward`, `zero_grad`, flat-parameter helpers for natural-gradient methods, `copy` and `soft_update` for target networks), the optimizers `Adam` (with optional gradient clipping by global norm) and `SGD`, and helpers for policies: `softmax`, `log_softmax`, `gaussian_log_prob` and `numerical_gradient`. Appendix D lists the rest of the library.

The calling convention follows the backward pass (C.2): `forward(x)` stores the activations; `backward(dy)` takes $\partial \mathcal{L}/\partial(\text{output})$, *adds* the parameter gradients into `net.grads` (so call `zero_grad`

first) and returns $\partial \mathcal{L}/\partial x$. Here is a gradient check, the first test to run on any new network code:

```
import numpy as np
from rlcourse.nn import MLP, numerical_gradient

rng = np.random.default_rng(0)
net = MLP([3, 16, 16, 2], activation='tanh', rng=0)
x, w = rng.normal(size=(8, 3)), rng.normal(size=(8, 2))
loss = lambda: float((net.forward(x) * w).sum())     # so dLoss/doutput = w

net.zero_grad()
net.forward(x)
net.backward(w)                                      # analytic gradients in net.grads
for p, g in zip(net.params, net.grads):
    num = numerical_gradient(loss, p)                # central differences
    print(np.abs(g - num).max() / np.abs(g).max())
```

The loss $\sum_{ij} w_{ij}\,\hat{y}_{ij}$ is chosen because its gradient with respect to the output is simply w. For this network, with 370 parameters, the largest relative difference between the analytic and the numerical gradients is below 10^{-9}, the level of round-off error in central differences. A bug in a backward pass typically produces relative errors of order 1.

A value-based update shows how the pieces fit together in reinforcement learning. For a batch of transitions with states `s`, actions `a`, rewards `r`, next states `s2` and termination flags `term`, one DQN-style step with the squared TD error is:

```
from rlcourse.nn import MLP, Adam

q = MLP([4, 64, 64, 2], rng=0)                       # a Q-network for CartPole
target = q.copy()
opt = Adam(q.params, lr=1e-3, max_grad_norm=10.0)
# ... for each batch (s, a, r, s2, term) from the replay buffer:
y = r + gamma * (1 - term) * target.forward(s2).max(axis=1)   # bootstrap targets
qs = q.forward(s)                                    # shape (batch, 2)
rows = np.arange(len(a))
grad = np.zeros_like(qs)
grad[rows, a] = (qs[rows, a] - y) / len(a)           # d(mean of 0.5 * TD error^2)/dQ
q.zero_grad()
q.backward(grad)
opt.step(q.grads)
target.soft_update(q, 0.005)                         # Polyak-averaged target network
```

Only the output of the action actually taken receives a gradient, the targets are computed with the target network and treated as constants (no gradient flows into `y`), and the termination flag, not truncation, decides whether to bootstrap (Section 23.2). Chapter 10 builds the full algorithm around this step.

C.8 Exercises

Exercise C.1 • Backpropagation with a ReLU

A network has one input, two ReLU hidden units and a linear output, with $W_1 = (1, -1)$, $b_1 = (-1, 1)$, $W_2 = (0.5, 1.5)^\top$ and $b_2 = 0.2$. For the input $x = 2$, the target $y = 1$ and the loss $\frac{1}{2}(\hat{y} - y)^2$, compute the forward pass, the loss, and the gradients with respect to all parameters and to the input. Which parameters receive no gradient, and why?

Exercise C.2 •• Softmax gradients

(a) For logits $z \in \mathbb{R}^K$ and a label y, show that the gradient of the cross-entropy $-\log \mathrm{softmax}(z)_y$ with respect to z is $\mathrm{softmax}(z) - e_y$. (b) For a softmax policy with logits $z(s)$, deduce $\nabla_z \log \pi(a \mid s) = e_a - \pi(\cdot \mid s)$, and write the REINFORCE update for the logits after an episode with return G in which action a was taken in state s. (c) Why does the gradient in (a) never exceed $\sqrt{2}$ in Euclidean norm?

Exercise C.3 •• The score of a Gaussian policy

A one-dimensional Gaussian policy has mean μ and log standard deviation $\rho = \log \sigma$. (a) Compute $\partial \log \pi(a)/\partial \mu$ and $\partial \log \pi(a)/\partial \rho$. (b) Show that both have expectation zero under $a \sim \pi$. (c) What happens to the first derivative as $\sigma \to 0$, and why do implementations bound ρ from below?

Exercise C.4 •• The size of an Adam step

(a) Show that Adam's first update of a parameter is $-\eta\, g_1/(|g_1| + \epsilon)$, so that it has size almost exactly η whatever the scale of g_1. (b) If all rewards of a task are multiplied by 100, how do the gradients of a value network's squared loss change, and how do the updates of SGD and of Adam change? (c) Explain why this makes Adam's learning rate easier to transfer between tasks, and name one situation in which this invariance is undesirable.

Exercise C.5 •• Why Glorot initialization

Consider a linear layer $z = aW$ with inputs a_i that are independent with mean zero and variance v, and weights W_{ij} independent of them with mean zero and variance s^2. (a) Show that $\mathrm{Var}(z_j) = n_{\text{in}}\, s^2 v$. (b) Show similarly that, in the backward pass, $\mathrm{Var}\big((gW^\top)_i\big) = n_{\text{out}}\, s^2\, \mathrm{Var}(g_j)$. (c) Explain why $s^2 = 2/(n_{\text{in}} + n_{\text{out}})$ is a compromise between preserving the variance in both passes, and what changes for ReLU activations.

Exercise C.6 • The Huber loss

The Huber loss with threshold 1 is $\ell(e) = \frac{1}{2}e^2$ for $|e| \le 1$ and $|e| - \frac{1}{2}$ otherwise. (a) Show that ℓ is continuously differentiable and that $\ell'(e) = \mathrm{clip}(e, -1, 1)$. (b) Explain why minimizing it

with TD errors e is not the same as clipping the rewards, and why it limits the damage of a single huge TD error.

Exercise C.7 • Momentum in the library

Write a class `Momentum` with the same interface as `rlcourse.nn.SGD` (a constructor taking the parameter list and a learning rate, and a method `step(grads)`) that implements SGD with momentum $\beta = 0.9$. Use it to fit $\sin 3x$ as in Figure C.1, and compare its mean squared error after 2,000 steps with that of SGD at the same learning rate.

APPENDIX D

The Lab Guide

Every chapter of this book comes with a lab: a short Python program that you run, read and change. The labs are built on `rlcourse`, a small library that implements the book's algorithms in plain NumPy, so that every line of an algorithm can be read and every number in the text can be reproduced. This appendix explains how to install and run them (Section D.1), how the library is organized (Section D.2), which labs exist (Section D.3), how the tests work (Section D.4), and how to move on to the libraries used in research and industry (Section D.5).

D.1 Setting up

The labs need Python 3.10 or later with NumPy and Matplotlib; `pytest` runs the tests. Nothing else is required. SciPy is optional for almost everything: it is imported only by the few functions that solve linear programs, in the modules of Chapter 3, Chapter 21 and Chapter 23. The labs and tests of Chapters 3, 4 and 23 skip their linear-programming checks when SciPy is missing; the lab and tests of Chapter 21 and the tests of Chapter 2 need it. To open the labs as notebooks you also need Jupyter with the Jupytext extension, or VS Code with its Python extension.

A clean setup in a virtual environment takes three commands, run from the course folder:

```
python3 -m venv .venv && source .venv/bin/activate      # Windows: .venv\Scripts\activate
python3 -m pip install numpy matplotlib pytest            # required
python3 -m pip install scipy jupyterlab jupytext          # optional
```

Run a lab from the course folder, or from anywhere, with its path:

```
python3 labs/ch06_td.py
```

Each lab adds the `labs/` folder to Python's search path at the top, so `import rlcourse` works wherever you start it. Labs print their results and save plots to `labs/out/`; they never open windows, so they also run on a server. Every lab finishes in under three minutes on a laptop CPU, most in well under a minute (Table D.2).

Labs as notebooks. The labs are written in the *percent* format: plain Python files in which a line `# %%` starts a code cell and `# %% [markdown]` starts a text cell. VS Code shows the cells and runs them one at a time. In Jupyter with Jupytext, a lab opens directly as a notebook; `jupytext --to notebook labs/ch06_td.py` converts it to an `.ipynb` file if you prefer. Working cell by cell is the best way to do the tasks: run a cell, read its output, change a parameter, run it again.

D.2 The `rlcourse` package

D.2.1 Design

The library is meant to be read. It has no dependencies beyond NumPy; it prefers loops that mirror the book's pseudocode to clever vectorization; its docstrings name the algorithm they implement ("Algorithm *Q-learning* in Chapter 6"); and its variable names follow the book's notation: `gamma`, `eta` (step size), `epsilon`, `delta` (TD error), `lam`, `pi[s, a]` (a policy as a matrix), `P[s, a, s2]` and `R[s, a]` (a model). Every function that uses randomness takes a `rng` or `seed` argument, so every run can be repeated exactly.

D.2.2 The core modules

Four modules are shared by all chapters:

- `rlcourse.envs`: the environments, all with the Gymnasium interface described below: the Trail (with an optional slip probability), grid worlds (including the Russell–Norvig 4×3 world), the cliff, the random walk, Gaussian and Bernoulli bandits and the 10-armed testbed, CartPole, Pendulum and Mountain Car.
- `rlcourse.mdp`: the class `FiniteMDP`, which stores a finite MDP as arrays and solves it exactly: `evaluate(pi)` solves the Bellman equation as a linear system, `value_iteration`, `policy_iteration` and `solve` find optimal values and policies, `discounted_visitation(pi)` computes d^π, and `rollout` samples episodes. Every tabular environment has a method `to_mdp(gamma)` that returns its `FiniteMDP`, so that a learning algorithm can be checked against the exact answer.
- `rlcourse.nn`: a multilayer perceptron with backpropagation, the Adam and SGD optimizers, and helpers for policies (softmax, log-softmax, Gaussian log-densities, numerical gradients); Section C.7 walks through it.
- `rlcourse.utils`: ε-greedy action selection, discounted returns, schedules, a replay buffer and running mean–variance statistics for normalization.

D.2.3 The interface

Environments follow Gymnasium (Towers et al. 2024), so code written for them carries over to real libraries:

```
from rlcourse.envs import Trail

env = Trail(slip=0.2)
s, info = env.reset(seed=0)
done = False
while not done:
    a = Trail.GO
    s, r, terminated, truncated, info = env.step(a)
    done = terminated or truncated
# the exact answer to compare with:
V_star, Q_star, pi_star = env.to_mdp(gamma=0.9).solve()
```

`terminated` means that a terminal state was reached, so the future is worth zero; `truncated` means that a time limit (the argument `max_steps` of every environment) cut the episode short, and the state reached still has a value. Mixing the two up is a classic bug (Section 23.2). Tabular environments number their states and actions from zero; the Trail's states are `Trail.START`, `MIDDLE`, `GOAL` and `END` (0 to 3) and its actions `Trail.EXIT` and `Trail.GO` (0 and 1). In a `FiniteMDP`, terminal states are absorbing with zero reward, so every formula of the book applies to them unchanged.

D.2.4 The chapter modules

Each chapter from Chapter 2 on adds one module (Chapter 23 adds two), listed in Table D.1. A chapter's module builds on the core modules and, in a few cases, on the module of an earlier chapter: `td` reuses a helper of `mc`, `ppo` builds on the policies and advantage estimates of `pg`, and `pg_theory` on the soft value functions of `continuous`.

Table D.1. The chapter modules of `rlcourse` (in `labs/rlcourse/`).

Module	Ch.	Contents
`bandits`	2	ε-greedy, UCB1, Thompson sampling, gradient bandits, LinUCB
`mdp_tools`	3	occupancy measures, advantages, performance difference
`dp_examples`	4	dynamic programming written out step by step
`mc`	5	Monte Carlo prediction and control, importance sampling
`td`	6	TD(0), SARSA, Q-learning, Expected SARSA, Double Q
`nstep`	7	n-step methods, λ-returns, eligibility traces
`planning`	8	Dyna, prioritized sweeping, RTDP, tree search
`approx`	9	features, semi-gradient TD, LSTD, gradient TD, FQI
`dqn`	10	DQN and its extensions
`pg`	11	REINFORCE, baselines, actor-critic, GAE
`ppo`	12	natural gradients, TRPO, PPO
`continuous`	13	DDPG, TD3, SAC
`pg_theory`	14	exact policy gradients and policy mirror descent
`model_based`	15	learned models, model-predictive control
`exploration`	16	optimism, posterior sampling, exploration bonuses
`offline`	17	off-policy evaluation and offline RL
`imitation`	18	behavior cloning, DAgger, inverse RL, preferences
`pomdp`	19	beliefs, planning in POMDPs, memory
`hierarchy`	20	options, successor representations, hindsight
`marl`	21	matrix and Markov games, regret minimization
`llm`	22	language-model RL in miniature
`stats`, `safe`	23	evaluation statistics; shaping, constraints, risk

Module	Ch.	Contents
`frontiers`	24	small experiments on generalization and plasticity

D.3 The labs

Table D.2 lists the labs. The run time is the wall-clock time of the whole script on a laptop CPU; the time needed to work through the tasks and the exercises is much longer, typically 30 to 60 minutes, as stated at the top of each lab.

Table D.2. The labs, in `labs/`. Run times were measured on a laptop CPU and vary with the machine.

Ch.	File	Topic	Time
1	`ch01_intro.py`	Meet the Trail	1 s
2	`ch02_bandits.py`	Bandits on the testbed	45 s
3	`ch03_mdp.py`	MDPs in code	8 s
4	`ch04_dp.py`	Dynamic programming from scratch	1 s
5	`ch05_mc.py`	Monte Carlo methods on the Trail and the 4x3 world	10 s
6	`ch06_td.py`	Temporal-difference learning	10 s
7	`ch07_nstep.py`	Multi-step returns and eligibility traces	5 s
8	`ch08_planning.py`	Dyna, prioritized sweeping, RTDP, tree search	15 s
9	`ch09_vfa.py`	Value function approximation	3 s
10	`ch10_dqn.py`	Deep Q-networks on CartPole	37 s
11	`ch11_pg.py`	Policy gradients from the Trail to CartPole	7 s
12	`ch12_ppo.py`	Natural gradients, trust regions and PPO	12 s
13	`ch13_continuous.py`	Continuous control with DDPG, TD3 and SAC	66 s
14	`ch14_pg_theory.py`	Why policy gradients find optimal policies	3 s
15	`ch15_model_based.py`	Learning models and planning with them	17 s
16	`ch16_exploration.py`	Deep exploration	34 s
17	`ch17_offline.py`	Learning and evaluating from a fixed data set	10 s
18	`ch18_imitation.py`	Imitation, inverse RL and preferences	3 s
19	`ch19_pomdp.py`	Beliefs, alpha-vectors and memory	12 s
20	`ch20_hierarchy.py`	Options, successor representations, hindsight	9 s
21	`ch21_marl.py`	Games, equilibria and learning among agents	16 s
22	`ch22_llm.py`	Preferences and RLHF in miniature	2 s
22	`ch22_rlvr.py`	Verifiable rewards in miniature	12 s
23	`ch23_practice.py`	Doing RL well	19 s
24	`ch24_frontiers.py`	Two frontiers in miniature	34 s

Ch.	File	Topic	Time

Every lab has the same shape: a goal and the sections it needs, a setup cell, three to seven tasks with code that runs as written, "what to look for" notes after each task, and "going further" ideas at the end, many of which are developed in the chapter's programming exercises. Plots are written to `labs/out/` with the lab's chapter number in the file name.

D.4 Running the tests

The folder `labs/tests/` contains one test file per chapter, `test_chNN_*.py`, plus `test_core_*.py` for the shared modules and `test_appC_nn.py` for Appendix C. Run them from the `labs/` folder:

```
cd labs
python3 -m pytest -q tests                      # everything
python3 -m pytest -q tests/test_ch06*.py        # one chapter
python3 -m pytest -q tests/test_core*.py        # the shared core
```

The tests check the library against exact answers wherever possible: the Trail's values ($V^{\pi}(\text{Start}) = 1.76$ for the uniform policy, $V^{*}(\text{Start}) = 3.24$), the optimal return of the cliff (-13), the true values of the random walk, analytic gradients against finite differences, the constrained optimum of Chapter 23 against a linear program. Where only a statistical statement is possible ("Q-learning finds the optimal policy", "a bootstrap interval covers the mean about 95% of the time"), the tests use fixed seeds and generous thresholds, so that they pass reliably and still fail if the algorithm breaks. Each chapter's tests run in under 30 seconds.

The tests are also a model for your own work. When you change an algorithm in a lab, or write a new one, add a test that compares it with a case you can solve exactly; Section 23.3 explains why this matters more in reinforcement learning than almost anywhere else.

D.5 Moving to real libraries

`rlcourse` is built for reading, not for speed or scale. Its networks run on the CPU in NumPy, its environments are small, and it implements each algorithm once, plainly. For research and applications, use the libraries that the community maintains. Because the course follows their conventions, moving is mostly a matter of renaming.

Environments: Gymnasium. Gymnasium (Towers et al. 2024), the successor of OpenAI Gym (Brockman et al. 2016), provides the classic-control tasks of the course (`CartPole-v1`, `Pendulum-v1`, `MountainCar-v0`), toy-text grid tasks such as the cliff, the Atari games, and MuJoCo robotics tasks, all with the same `reset` and `step` interface; `gym.make(name)` creates one. Its *wrappers* add time limits, observation and reward normalization, episode statistics and frame stacking, and its vector environments run many copies at once. Read Section 23.2 before using them: the reset convention of vector environments differs between versions.

Networks: PyTorch or JAX. PyTorch (Paszke et al. 2019) computes gradients automatically, so the `backward` pass of `rlcourse.nn` becomes a single call. The DQN-style update of Section C.7 reads, in PyTorch:

```
import copy

import torch
import torch.nn as nn

q = nn.Sequential(nn.Linear(4, 64), nn.Tanh(),
                  nn.Linear(64, 64), nn.Tanh(), nn.Linear(64, 2))
target = copy.deepcopy(q)
opt = torch.optim.Adam(q.parameters(), lr=1e-3)
# ... for each batch of tensors (s, a, r, s2, term):
with torch.no_grad():                                    # targets are constants
    y = r + gamma * (1 - term) * target(s2).max(dim=1).values
loss = 0.5 * ((q(s).gather(1, a[:, None]).squeeze(1) - y) ** 2).mean()
opt.zero_grad()
loss.backward()                                          # replaces net.backward(grad)
torch.nn.utils.clip_grad_norm_(q.parameters(), 10.0)
opt.step()
```

JAX (Bradbury et al. 2018) takes a functional approach: gradients are functions (`jax.grad`), and whole training loops can be compiled for accelerators, which is what makes the GPU-based simulation discussed in Chapter 23 fast.

Algorithms: Stable-Baselines3 and CleanRL. Stable-Baselines3 (Raffin et al. 2021) offers tested implementations of PPO, A2C, DQN, SAC, TD3 and DDPG behind a common interface; training an agent takes two lines, for example `model = PPO('MlpPolicy', 'CartPole-v1', seed=0)` followed by `model.learn(total_timesteps=100_000)`. CleanRL (Huang, Dossa, Ye, et al. 2022) takes the opposite approach: each algorithm is a single, self-contained script (for example `ppo.py`, `dqn.py` and `sac_continuous_action.py`) that you read and modify, much like the labs of this book but at full scale. Use Stable-Baselines3 to get reliable baselines quickly, and CleanRL when you want to change an algorithm's internals.

Table D.3 maps the main names of `rlcourse` to their counterparts.

Table D.3. From `rlcourse` to Gymnasium, PyTorch, Stable-Baselines3 (SB3) and CleanRL.

`rlcourse`	Real libraries
`envs.CartPole()`, `Pendulum()`	`gymnasium.make('CartPole-v1')`, `'Pendulum-v1'`
`Env(max_steps=...)`	the `TimeLimit` wrapper (set by `gymnasium.make`)
`utils.RunningMeanStd`	`NormalizeObservation`, `NormalizeReward`; SB3 `VecNormalize`
`utils.ReplayBuffer`	SB3 `ReplayBuffer` (also used by CleanRL)
`nn.MLP(sizes)`	`torch.nn.Sequential` of `Linear` and activation layers

`rlcourse`	Real libraries
`net.backward(dy)`, `opt.step(net.grads)`	`loss.backward()`, `optimizer.step()`
`nn.Adam(..., max_grad_norm=c)`	`torch.optim.Adam` with `clip_grad_norm_`
`net.copy()`, `net.soft_update(q, tau)`	`copy.deepcopy`, a loop over parameters
`dqn`, `ppo`, `continuous`	SB3 `DQN`, `PPO`, `SAC`, `TD3`; CleanRL scripts
`stats.iqm`, `stratified_bootstrap_ci`	the `rliable` library of R. Agarwal et al. (2021)
`mdp.FiniteMDP`	no standard equivalent: keep it for exact checks

Two habits carry over unchanged. Keep a small problem with a known answer next to every large experiment, and evaluate with many seeds and honest intervals (Section 23.4); `rlcourse.stats` works on the scores of any library.

D.6 Exercises

Exercise D.1 • A new environment

Write a class for the *two-armed corridor*: states $0, 1, \ldots, 5$, start in state 0, actions left and right (moving one cell, bounded at 0), reward $+1$ for reaching state 5, which is terminal, and 0 otherwise. Give it the interface of `rlcourse.envs` (`reset(seed)`, `step(a)` with the five return values, a `max_steps` time limit) and a method `to_mdp(gamma)`. Write a test that checks `to_mdp(0.9).solve()` against the values you compute by hand.

Exercise D.2 •• From the course to Stable-Baselines3

Install Gymnasium and Stable-Baselines3, train PPO on `CartPole-v1` with ten seeds for 50,000 steps each, and evaluate each trained agent on 20 episodes. Report the mean return with a t interval and the interquartile mean with a bootstrap interval, using `rlcourse.stats`. Compare with the PPO of Lab 12.1, and list the differences in default settings that might explain any difference in results.

Exercise D.3 •• Porting a network to PyTorch

Rewrite the gradient check of Section C.7 in PyTorch: build the same $[3, 16, 16, 2]$ tanh network, copy the weights from an `rlcourse.nn.MLP` into it (mind the orientation of the weight matrices), and check that the outputs and the gradients agree to about 10^{-6} in single precision.

References

Abbas, Zaheer, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C. Machado. 2023. "Loss of Plasticity in Continual Deep Reinforcement Learning." *Conference on Lifelong Learning Agents*.

Abbasi-Yadkori, Yasin, Dávid Pál, and Csaba Szepesvári. 2011. "Improved Algorithms for Linear Stochastic Bandits." *Advances in Neural Information Processing Systems* 24.

Abbeel, Pieter, and Andrew Y. Ng. 2004. "Apprenticeship Learning via Inverse Reinforcement Learning." *Proceedings of the 21st International Conference on Machine Learning*.

Abdolmaleki, Abbas, Jost Tobias Springenberg, Yuval Tassa, Rémi Munos, Nicolas Heess, and Martin Riedmiller. 2018. "Maximum a Posteriori Policy Optimisation." *International Conference on Learning Representations*.

Abel, David, André Barreto, Benjamin Van Roy, Doina Precup, Hado van Hasselt, and Satinder Singh. 2023. "A Definition of Continual Reinforcement Learning." *Advances in Neural Information Processing Systems* 36.

Abel, David, Will Dabney, Anna Harutyunyan, et al. 2021. "On the Expressivity of Markov Reward." *Advances in Neural Information Processing Systems* 34.

Achiam, Joshua, David Held, Aviv Tamar, and Pieter Abbeel. 2017. "Constrained Policy Optimization." *Proceedings of the 34th International Conference on Machine Learning*, 22–31.

Adaptive Agent Team, Jakob Bauer, Kate Baumli, et al. 2023. "Human-Timescale Adaptation in an Open-Ended Task Space." *Proceedings of the 40th International Conference on Machine Learning*.

Agarwal, Alekh, Mikael Henaff, Sham Kakade, and Wen Sun. 2020. "PC-PG: Policy Cover Directed Exploration for Provable Policy Gradient Learning." *Advances in Neural Information Processing Systems* 33.

Agarwal, Alekh, Nan Jiang, Sham M. Kakade, and Wen Sun. 2022. *Reinforcement Learning: Theory and Algorithms*. Book draft, https://rltheorybook.github.io.

Agarwal, Alekh, Sham M. Kakade, Jason D. Lee, and Gaurav Mahajan. 2021. "On the Theory of Policy Gradient Methods: Optimality, Approximation, and Distribution Shift." *Journal of Machine Learning Research* 22 (98): 1–76.

Agarwal, Alekh, Sham Kakade, and Lin F. Yang. 2020. "Model-Based Reinforcement Learning with a Generative Model Is Minimax Optimal." *Proceedings of the 33rd Conference on Learning Theory*.

Agarwal, Rishabh, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. 2021. "Deep Reinforcement Learning at the Edge of the Statistical Precipice." *Advances in Neural Information Processing Systems* 34.

Agrawal, Shipra, and Navin Goyal. 2012. "Analysis of Thompson Sampling for the Multi-Armed Bandit Problem." *Proceedings of the 25th Annual Conference on Learning Theory*.

Agrawal, Shipra, and Navin Goyal. 2013. "Thompson Sampling for Contextual Bandits with Linear Payoffs." *Proceedings of the 30th International Conference on Machine Learning*.

Ahmadian, Arash, Chris Cremer, Matthias Gallé, et al. 2024. "Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs." *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

Ahmed, Zafarali, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. 2019. "Understanding the Impact of Entropy on Policy Optimization." *Proceedings of the 36th International Conference on Machine Learning*.

Ahn, Michael, Anthony Brohan, Noah Brown, et al. 2022. "Do as I Can, Not as I Say: Grounding Language in Robotic Affordances." *Conference on Robot Learning*.

Albrecht, Stefano V., Filippos Christianos, and Lukas Schäfer. 2024. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press.

Alshiekh, Mohammed, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. "Safe Reinforcement Learning via Shielding." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Altman, Eitan. 1999. *Constrained Markov Decision Processes*. Chapman; Hall/CRC.

Amari, Shun-ichi. 1998. "Natural Gradient Works Efficiently in Learning." *Neural Computation* 10 (2): 251–76.

Amodei, Dario, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. "Concrete Problems in AI Safety." *arXiv Preprint arXiv:1606.06565*.

Andrychowicz, Marcin, Anton Raichuk, Piotr Stańczyk, et al. 2021. "What Matters for on-Policy Deep Actor-Critic Methods? A Large-Scale Study." *International Conference on Learning Representations*.

Andrychowicz, Marcin, Filip Wolski, Alex Ray, et al. 2017. "Hindsight Experience Replay." *Advances in Neural Information Processing Systems* 30.

Anthony, Thomas, Zheng Tian, and David Barber. 2017. "Thinking Fast and Slow with Deep Learning and Tree Search." *Advances in Neural Information Processing Systems* 30.

Antonoglou, Ioannis, Julian Schrittwieser, Sherjil Ozair, Thomas Hubert, and David Silver. 2022. "Planning in Stochastic Environments with a Learned Model." *International Conference on Learning Representations*.

Ash, Jordan T., and Ryan P. Adams. 2020. "On Warm-Starting Neural Network Training." *Advances in Neural Information Processing Systems* 33.

Åström, Karl Johan. 1965. "Optimal Control of Markov Processes with Incomplete State Information." *Journal of Mathematical Analysis and Applications* 10 (1): 174–205.

Audibert, Jean-Yves, and Sébastien Bubeck. 2009. "Minimax Policies for Adversarial and Stochastic Bandits." *Proceedings of the 22nd Annual Conference on Learning Theory*.

Auer, Peter, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. "Finite-Time Analysis of the Multiarmed Bandit Problem." *Machine Learning* 47 (2–3): 235–56.

Auer, Peter, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2002. "The Nonstochastic Multi-armed Bandit Problem." *SIAM Journal on Computing* 32 (1): 48–77.

Aumann, Robert J. 1974. "Subjectivity and Correlation in Randomized Strategies." *Journal of Mathematical Economics* 1 (1): 67–96.

Azar, Mohammad Gheshlaghi, Zhaohan Daniel Guo, Bilal Piot, et al. 2024. "A General Theoretical Paradigm to Understand Learning from Human Preferences." *Proceedings of the 27th International Conference on Artificial Intelligence and Statistics*.

Azar, Mohammad Gheshlaghi, Rémi Munos, and Hilbert J. Kappen. 2013. "Minimax PAC Bounds on the Sample Complexity of Reinforcement Learning with a Generative Model." *Machine Learning* 91 (3): 325–49.

Azar, Mohammad Gheshlaghi, Ian Osband, and Rémi Munos. 2017. "Minimax Regret Bounds for Reinforcement Learning." *Proceedings of the 34th International Conference on Machine Learning*.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. "Layer Normalization." *arXiv Preprint arXiv:1607.06450*.

Bacon, Pierre-Luc, Jean Harb, and Doina Precup. 2017. "The Option-Critic Architecture." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Badia, Adrià Puigdomènech, Bilal Piot, Steven Kapturowski, et al. 2020. "Agent57: Outperforming the Atari Human Benchmark." *Proceedings of the 37th International Conference on Machine Learning*.

Badia, Adrià Puigdomènech, Pablo Sprechmann, Alex Vitvitskyi, et al. 2020. "Never Give up: Learning Directed Exploration Strategies." *International Conference on Learning Representations*.

Bai, Yu, and Chi Jin. 2020. "Provable Self-Play Algorithms for Competitive Reinforcement Learning." *Proceedings of the 37th International Conference on Machine Learning*.

Bai, Yuntao, Andy Jones, Kamal Ndousse, et al. 2022. "Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback." *arXiv Preprint arXiv:2204.05862*.

Bai, Yuntao, Saurav Kadavath, Sandipan Kundu, et al. 2022. "Constitutional AI: Harmlessness from AI

Feedback." *arXiv Preprint arXiv:2212.08073*.

Bailey, James P., and Georgios Piliouras. 2018. "Multiplicative Weights Update in Zero-Sum Games." *Proceedings of the 2018 ACM Conference on Economics and Computation*.

Baird, Leemon. 1995. "Residual Algorithms: Reinforcement Learning with Function Approximation." *Proceedings of the Twelfth International Conference on Machine Learning*, 30–37.

Baker, Bowen, Joost Huizinga, Leo Gao, et al. 2025. "Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation." *arXiv Preprint*.

Bakker, Bram. 2001. "Reinforcement Learning with Long Short-Term Memory." *Advances in Neural Information Processing Systems* 14.

Ball, Philip J., Laura Smith, Ilya Kostrikov, and Sergey Levine. 2023. "Efficient Online Reinforcement Learning with Offline Data." *Proceedings of the 40th International Conference on Machine Learning*.

Banach, Stefan. 1922. "Sur Les Opérations Dans Les Ensembles Abstraits Et Leur Application Aux équations Intégrales." *Fundamenta Mathematicae* 3: 133–81.

Bard, Nolan, Jakob N. Foerster, Sarath Chandar, et al. 2020. "The Hanabi Challenge: A New Frontier for AI Research." *Artificial Intelligence* 280: 103216.

Barreto, André, Will Dabney, Rémi Munos, et al. 2017. "Successor Features for Transfer in Reinforcement Learning." *Advances in Neural Information Processing Systems* 30.

Barreto, André, Shaobo Hou, Diana Borsa, David Silver, and Doina Precup. 2020. "Fast Reinforcement Learning with Generalized Policy Updates." *Proceedings of the National Academy of Sciences* 117 (48): 30079–87.

Barth-Maron, Gabriel, Matthew W. Hoffman, David Budden, et al. 2018. "Distributed Distributional Deterministic Policy Gradients." *International Conference on Learning Representations*.

Barto, Andrew G., Steven J. Bradtke, and Satinder P. Singh. 1995. "Learning to Act Using Real-Time Dynamic Programming." *Artificial Intelligence* 72 (1–2): 81–138.

Barto, Andrew G., and Michael Duff. 1994. "Monte Carlo Matrix Inversion and Reinforcement Learning." *Advances in Neural Information Processing Systems* 6.

Barto, Andrew G., and Sridhar Mahadevan. 2003. "Recent Advances in Hierarchical Reinforcement Learning." *Discrete Event Dynamic Systems* 13.

Barto, Andrew G., Richard S. Sutton, and Charles W. Anderson. 1983. "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems." *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13 (5): 834–46.

Bastani, Hamsa, Mohsen Bayati, and Khashayar Khosravi. 2021. "Mostly Exploration-Free Algorithms for Contextual Bandits." *Management Science* 67 (3): 1329–49.

Baxter, Jonathan, and Peter L. Bartlett. 2001. "Infinite-Horizon Policy-Gradient Estimation." *Journal of Artificial Intelligence Research* 15: 319–50.

Beck, Jacob, Risto Vuorio, Evan Zheran Liu, et al. 2023. "A Survey of Meta-Reinforcement Learning." *arXiv Preprint arXiv:2301.08028*.

Beirami, Ahmad, Alekh Agarwal, Jonathan Berant, et al. 2024. "Theoretical Guarantees on the Best-of-*n* Alignment Policy." *arXiv Preprint*.

Bellemare, Marc G., Salvatore Candido, Pablo Samuel Castro, et al. 2020. "Autonomous Navigation of Stratospheric Balloons Using Reinforcement Learning." *Nature* 588 (7836): 77–82.

Bellemare, Marc G., Will Dabney, Robert Dadashi, et al. 2019. "A Geometric Perspective on Optimal Representations for Reinforcement Learning." *Advances in Neural Information Processing Systems* 32.

Bellemare, Marc G., Will Dabney, and Rémi Munos. 2017. "A Distributional Perspective on Reinforcement Learning." *Proceedings of the 34th International Conference on Machine Learning*, 449–58.

Bellemare, Marc G., Will Dabney, and Mark Rowland. 2023. *Distributional Reinforcement Learning*. MIT Press.

Bellemare, Marc G., Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. "The Arcade Learning Environment: An Evaluation Platform for General Agents." *Journal of Artificial Intelligence Research* 47: 253–79.

Bellemare, Marc G., Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. 2016. "Unifying Count-Based Exploration and Intrinsic Motivation." *Advances in Neural Information Processing Systems* 29.

Bellman, Richard. 1957a. "A Markovian Decision Process." *Journal of Mathematics and Mechanics* 6 (5): 679–84.

Bellman, Richard. 1957b. *Dynamic Programming*. Princeton University Press.

Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks." *Advances in Neural Information Processing Systems* 28.

Benveniste, Albert, Michel Métivier, and Pierre Priouret. 1990. *Adaptive Algorithms and Stochastic Approximations*. Springer.

Bergstra, James, and Yoshua Bengio. 2012. "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research* 13: 281–305.

Berner, Christopher, Greg Brockman, Brooke Chan, et al. 2019. "Dota 2 with Large Scale Deep Reinforcement Learning." *arXiv Preprint arXiv:1912.06680*.

Bernstein, Daniel S., Robert Givan, Neil Immerman, and Shlomo Zilberstein. 2002. "The Complexity

of Decentralized Control of Markov Decision Processes." *Mathematics of Operations Research* 27 (4): 819–40.

Bertsekas, Dimitri P. 2012. *Dynamic Programming and Optimal Control, Volume II: Approximate Dynamic Programming*. 4th ed. Athena Scientific.

Bertsekas, Dimitri P. 2017. *Dynamic Programming and Optimal Control, Volume I*. 4th ed. Athena Scientific.

Bertsekas, Dimitri P. 2019. *Reinforcement Learning and Optimal Control*. Athena Scientific.

Bertsekas, Dimitri P., and John N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Bhandari, Jalaj, and Daniel Russo. 2021. "On the Linear Convergence of Policy Gradient Methods for Finite MDPs." *Proceedings of the 24th International Conference on Artificial Intelligence and Statistics*.

Bhandari, Jalaj, and Daniel Russo. 2024. "Global Optimality Guarantees for Policy Gradient Methods." *Operations Research*.

Bhandari, Jalaj, Daniel Russo, and Raghav Singal. 2018. "A Finite Time Analysis of Temporal Difference Learning with Linear Function Approximation." *Proceedings of the 31st Conference on Learning Theory*.

Bhatnagar, Shalabh, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. 2009. "Natural Actor-Critic Algorithms." *Automatica* 45 (11): 2471–82.

Bhatt, Aditya, Daniel Palenicek, Boris Belousov, et al. 2024. "CrossQ: Batch Normalization in Deep Reinforcement Learning for Greater Sample Efficiency and Simplicity." *International Conference on Learning Representations*.

Blackwell, David. 1962. "Discrete Dynamic Programming." *The Annals of Mathematical Statistics* 33 (2): 719–26.

Borkar, Vivek S. 1997. "Stochastic Approximation with Two Time Scales." *Systems & Control Letters* 29 (5): 291–94.

Borkar, Vivek S. 2008. *Stochastic Approximation: A Dynamical Systems Viewpoint*. Cambridge University Press.

Borkar, Vivek S., and Sean P. Meyn. 2000. "The O.D.E. Method for Convergence of Stochastic Approximation and Reinforcement Learning." *SIAM Journal on Control and Optimization* 38 (2): 447–69.

Boucheron, Stéphane, Gábor Lugosi, and Pascal Massart. 2013. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press.

Bowling, Michael, Neil Burch, Michael Johanson, and Oskari Tammelin. 2015. "Heads-up Limit Hold'em Poker Is Solved." *Science* 347 (6218): 145–49.

Bowling, Michael, John D. Martin, David Abel, and Will Dabney. 2023. "Settling the Reward Hypothesis." *Proceedings of the 40th International Conference on Machine Learning*.

Bowling, Michael, and Manuela Veloso. 2002. "Multiagent Learning Using a Variable Learning Rate." *Artificial Intelligence* 136 (2): 215–50.

Boyan, Justin A. 2002. "Technical Update: Least-Squares Temporal Difference Learning." *Machine Learning* 49: 233–46.

Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press.

Bradbury, James et al. 2018. *JAX: Composable Transformations of Python+NumPy Programs*. Https://github.com/google/jax.

Bradley, Ralph Allan, and Milton E. Terry. 1952. "Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons." *Biometrika* 39 (3/4): 324–45.

Bradtke, Steven J., and Andrew G. Barto. 1996. "Linear Least-Squares Algorithms for Temporal Difference Learning." *Machine Learning* 22: 33–57.

Bradtke, Steven J., and Michael O. Duff. 1995. "Reinforcement Learning Methods for Continuous-Time Markov Decision Problems." *Advances in Neural Information Processing Systems* 7.

Brafman, Ronen I., and Moshe Tennenholtz. 2002. "R-max – a General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning." *Journal of Machine Learning Research* 3: 213–31.

Brandfonbrener, David, Alberto Bietti, Jacob Buckman, Romain Laroche, and Joan Bruna. 2022. "When Does Return-Conditioned Supervised Learning Work for Offline Reinforcement Learning?" *Advances in Neural Information Processing Systems* 35.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, et al. 2016. "OpenAI Gym." *arXiv Preprint arXiv:1606.01540*.

Brohan, Anthony, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, et al. 2023. "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control." *arXiv Preprint arXiv:2307.15818*.

Brohan, Anthony, Noah Brown, Justice Carbajal, Yevgen Chebotar, et al. 2023. "RT-1: Robotics Transformer for Real-World Control at Scale." *Robotics: Science and Systems*.

Brown, George W. 1951. "Iterative Solution of Games by Fictitious Play." In *Activity Analysis of Production and Allocation*. Wiley.

Brown, Noam, Anton Bakhtin, Adam Lerer, and Qucheng Gong. 2020. "Combining Deep Reinforcement Learning and Search for Imperfect-Information Games." *Advances in Neural Information Processing Systems* 33.

Brown, Noam, Adam Lerer, Sam Gross, and Tuomas Sandholm. 2019. "Deep Counterfactual Regret

Minimization." *Proceedings of the 36th International Conference on Machine Learning*.

Brown, Noam, and Tuomas Sandholm. 2018. "Superhuman AI for Heads-up No-Limit Poker: Libratus Beats Top Professionals." *Science* 359 (6374): 418–24.

Brown, Noam, and Tuomas Sandholm. 2019a. "Solving Imperfect-Information Games via Discounted Regret Minimization." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Brown, Noam, and Tuomas Sandholm. 2019b. "Superhuman AI for Multiplayer Poker." *Science* 365 (6456): 885–90.

Browne, Cameron B., Edward Powley, Daniel Whitehouse, et al. 2012. "A Survey of Monte Carlo Tree Search Methods." *IEEE Transactions on Computational Intelligence and AI in Games* 4 (1): 1–43.

Bubeck, Sébastien, and Nicolò Cesa-Bianchi. 2012. "Regret Analysis of Stochastic and Nonstochastic Multi-Armed Bandit Problems." *Foundations and Trends in Machine Learning* 5 (1): 1–122.

Bubeck, Sébastien, Rémi Munos, and Gilles Stoltz. 2009. "Pure Exploration in Multi-Armed Bandits Problems." *Algorithmic Learning Theory (ALT 2009)*.

Buckman, Jacob, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. 2018. "Sample-Efficient Reinforcement Learning with Stochastic Ensemble Value Expansion." *Advances in Neural Information Processing Systems* 31.

Burda, Yuri, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. 2019. "Large-Scale Study of Curiosity-Driven Learning." *International Conference on Learning Representations*.

Burda, Yuri, Harrison Edwards, Amos Storkey, and Oleg Klimov. 2019. "Exploration by Random Network Distillation." *International Conference on Learning Representations*.

Burnetas, Apostolos N., and Michael N. Katehakis. 1996. "Optimal Adaptive Policies for Sequential Allocation Problems." *Advances in Applied Mathematics* 17 (2): 122–42.

Burns, Collin, Pavel Izmailov, Jan Hendrik Kirchner, et al. 2024. "Weak-to-Strong Generalization: Eliciting Strong Capabilities with Weak Supervision." *Proceedings of the 41st International Conference on Machine Learning*.

Cai, Qi, Zhuoran Yang, Chi Jin, and Zhaoran Wang. 2020. "Provably Efficient Exploration in Policy Optimization." *Proceedings of the 37th International Conference on Machine Learning*.

Casper, Stephen, Xander Davies, Claudia Shi, et al. 2023. "Open Problems and Fundamental Limitations of Reinforcement Learning from Human Feedback." *Transactions on Machine Learning Research*.

Cassandra, Anthony, Michael L. Littman, and Nevin L. Zhang. 1997. "Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes." *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*.

Cen, Shicong, Chen Cheng, Yuxin Chen, Yuting Wei, and Yuejie Chi. 2022. "Fast Global Convergence of Natural Policy Gradient Methods with Entropy Regularization." *Operations Research* 70 (4).

Cen, Shicong, Yuting Wei, and Yuejie Chi. 2021. "Fast Policy Extragradient Methods for Competitive Games with Entropy Regularization." *Advances in Neural Information Processing Systems* 34.

Cesa-Bianchi, Nicolò, and Gábor Lugosi. 2006. *Prediction, Learning, and Games*. Cambridge University Press.

Chan, Stephanie C. Y., Samuel Fishman, John Canny, Anoop Korattikara, and Sergio Guadarrama. 2020. "Measuring the Reliability of Reinforcement Learning Algorithms." *International Conference on Learning Representations*.

Chapelle, Olivier, and Lihong Li. 2011. "An Empirical Evaluation of Thompson Sampling." *Advances in Neural Information Processing Systems* 24.

Chen, Jinglin, and Nan Jiang. 2019. "Information-Theoretic Considerations in Batch Reinforcement Learning." *Proceedings of the 36th International Conference on Machine Learning*.

Chen, Lili, Kevin Lu, Aravind Rajeswaran, et al. 2021. "Decision Transformer: Reinforcement Learning via Sequence Modeling." *Advances in Neural Information Processing Systems* 34.

Chen, Mark, Jerry Tworek, Heewoo Jun, et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv Preprint arXiv:2107.03374*.

Chen, Xi, Xiaotie Deng, and Shang-Hua Teng. 2009. "Settling the Complexity of Computing Two-Player Nash Equilibria." *Journal of the ACM* 56 (3).

Chen, Xinyue, Che Wang, Zijian Zhou, and Keith W. Ross. 2021. "Randomized Ensembled Double Q-Learning: Learning Fast Without a Model." *International Conference on Learning Representations*.

Cheng, Chung-Kuan, Andrew B. Kahng, Sayak Kundu, Yucheng Wang, and Zhiang Wang. 2023. "Assessment of Reinforcement Learning for Macro Placement." *Proceedings of the 2023 International Symposium on Physical Design*.

Chi, Cheng, Siyuan Feng, Yilun Du, et al. 2023. "Diffusion Policy: Visuomotor Policy Learning via Action Diffusion." *Robotics: Science and Systems*.

Cho, Kyunghyun, Bart van Merriënboer, Caglar Gulcehre, et al. 2014. "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation." *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*.

Chow, Yinlam, Mohammad Ghavamzadeh, Lucas Janson, and Marco Pavone. 2018. "Risk-Constrained Reinforcement Learning with Percentile Risk Criteria." *Journal of Machine Learning Research* 18.

Christiano, Paul F., Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. "Deep Reinforcement Learning from Human Preferences." *Advances in Neural Information Processing Systems* 30.

Christodoulou, Petros. 2019. "Soft Actor-Critic for Discrete Action Settings." *arXiv Preprint arXiv:1910.07207*.

Chu, Wei, Lihong Li, Lev Reyzin, and Robert E. Schapire. 2011. "Contextual Bandits with Linear Payoff Functions." *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*.

Chua, Kurtland, Roberto Calandra, Rowan McAllister, and Sergey Levine. 2018. "Deep Reinforcement Learning in a Handful of Trials Using Probabilistic Dynamics Models." *Advances in Neural Information Processing Systems* 31.

Clark, Jack, and Dario Amodei. 2016. *Faulty Reward Functions in the Wild*. OpenAI Blog.

Claus, Caroline, and Craig Boutilier. 1998. "The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems." *Proceedings of the Fifteenth National Conference on Artificial Intelligence*.

Cobbe, Karl, Christopher Hesse, Jacob Hilton, and John Schulman. 2020. "Leveraging Procedural Generation to Benchmark Reinforcement Learning." *Proceedings of the 37th International Conference on Machine Learning*.

Cobbe, Karl, Jacob Hilton, Oleg Klimov, and John Schulman. 2021. "Phasic Policy Gradient." *Proceedings of the 38th International Conference on Machine Learning*.

Cobbe, Karl, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. 2019. "Quantifying Generalization in Reinforcement Learning." *Proceedings of the 36th International Conference on Machine Learning*.

Cobbe, Karl, Vineet Kosaraju, Mohammad Bavarian, et al. 2021. "Training Verifiers to Solve Math Word Problems." *arXiv Preprint arXiv:2110.14168*.

Colas, Cédric, Olivier Sigaud, and Pierre-Yves Oudeyer. 2018. "How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments." *arXiv Preprint arXiv:1806.08295*.

Coquelin, Pierre-Arnaud, and Rémi Munos. 2007. "Bandit Algorithms for Tree Search." *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*.

Coste, Thomas, Usman Anwar, Robert Kirk, and David Krueger. 2024. "Reward Model Ensembles Help Mitigate Overoptimization." *International Conference on Learning Representations*.

Coulom, Rémi. 2006. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search." *Computers and Games*.

Cui, Ganqu, Yuchen Zhang, Jiacheng Chen, et al. 2025. "The Entropy Mechanism of Reinforcement Learning for Reasoning Language Models." *arXiv Preprint*.

Cybenko, George. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals and Systems* 2 (4): 303–14.

d'Epenoux, F. 1963. "A Probabilistic Production and Inventory Problem." *Management Science* 10 (1):

98–108.

D'Oro, Pierluca, Max Schwarzer, Evgenii Nikishin, Pierre-Luc Bacon, Marc G. Bellemare, and Aaron Courville. 2023. "Sample-Efficient Reinforcement Learning by Breaking the Replay Ratio Barrier." *International Conference on Learning Representations*.

Dabney, Will, Zeb Kurth-Nelson, Naoshige Uchida, et al. 2020. "A Distributional Code for Value in Dopamine-Based Reinforcement Learning." *Nature* 577: 671–75.

Dabney, Will, Georg Ostrovski, David Silver, and Rémi Munos. 2018. "Implicit Quantile Networks for Distributional Reinforcement Learning." *Proceedings of the 35th International Conference on Machine Learning*.

Dabney, Will, Mark Rowland, Marc G. Bellemare, and Rémi Munos. 2018. "Distributional Reinforcement Learning with Quantile Regression." *Proceedings of the AAAI Conference on Artificial Intelligence* 32.

Dalal, Gal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. 2018. "Safe Exploration in Continuous Action Spaces." *arXiv Preprint arXiv:1801.08757*.

Dani, Varsha, Thomas P. Hayes, and Sham M. Kakade. 2008. "Stochastic Linear Optimization Under Bandit Feedback." *Proceedings of the 21st Annual Conference on Learning Theory*.

Dann, Christoph, Tor Lattimore, and Emma Brunskill. 2017. "Unifying PAC and Regret: Uniform PAC Bounds for Episodic Reinforcement Learning." *Advances in Neural Information Processing Systems* 30.

Dann, Christoph, Gerhard Neumann, and Jan Peters. 2014. "Policy Evaluation with Temporal Differences: A Survey and Comparison." *Journal of Machine Learning Research* 15: 809–83.

Daskalakis, Constantinos, Dylan J. Foster, and Noah Golowich. 2020. "Independent Policy Gradient Methods for Competitive Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Daskalakis, Constantinos, Paul W. Goldberg, and Christos H. Papadimitriou. 2009. "The Complexity of Computing a Nash Equilibrium." *SIAM Journal on Computing* 39 (1): 195–259.

Daskalakis, Constantinos, Noah Golowich, and Kaiqing Zhang. 2023. "The Complexity of Markov Equilibrium in Stochastic Games." *Proceedings of the Thirty Sixth Conference on Learning Theory*.

Daskalakis, Constantinos, Andrew Ilyas, Vasilis Syrgkanis, and Haoyang Zeng. 2018. "Training GANs with Optimism." *International Conference on Learning Representations*.

Dayan, Peter. 1992. "The Convergence of TD(λ) for General λ." *Machine Learning* 8 (3–4): 341–62.

Dayan, Peter. 1993. "Improving Generalization for Temporal Difference Learning: The Successor Representation." *Neural Computation* 5 (4): 613–24.

Dayan, Peter, and Geoffrey E. Hinton. 1993. "Feudal Reinforcement Learning." *Advances in Neural*

Information Processing Systems 5.

De Asis, Kristopher, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton. 2018. "Multi-Step Reinforcement Learning: A Unifying Algorithm." *Proceedings of the AAAI Conference on Artificial Intelligence* 32.

de Boer, Pieter-Tjerk, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. 2005. "A Tutorial on the Cross-Entropy Method." *Annals of Operations Research* 134 (1): 19–67.

de Farias, Daniela Pucci, and Benjamin Van Roy. 2003. "The Linear Programming Approach to Approximate Dynamic Programming." *Operations Research* 51 (6): 850–65.

Degrave, Jonas, Federico Felici, Jonas Buchli, et al. 2022. "Magnetic Control of Tokamak Plasmas Through Deep Reinforcement Learning." *Nature* 602 (7897): 414–19.

Degris, Thomas, Martha White, and Richard S. Sutton. 2012. "Off-Policy Actor-Critic." *Proceedings of the 29th International Conference on Machine Learning*.

Deisenroth, Marc Peter, and Carl Edward Rasmussen. 2011. "PILCO: A Model-Based and Data-Efficient Approach to Policy Search." *Proceedings of the 28th International Conference on Machine Learning*.

Denardo, Eric V. 1967. "Contraction Mappings in the Theory Underlying Dynamic Programming." *SIAM Review* 9 (2): 165–77.

Dietterich, Thomas G. 2000. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition." *Journal of Artificial Intelligence Research* 13: 227–303.

Ding, Dongsheng, Kaiqing Zhang, Tamer Başar, and Mihailo R. Jovanović. 2020. "Natural Policy Gradient Primal-Dual Method for Constrained Markov Decision Processes." *Advances in Neural Information Processing Systems* 33.

Dohare, Shibhansh, J. Fernando Hernandez-Garcia, Qingfeng Lan, Parash Rahman, A. Rupam Mahmood, and Richard S. Sutton. 2024. "Loss of Plasticity in Deep Continual Learning." *Nature*.

Dolan, Elizabeth D., and Jorge J. Moré. 2002. "Benchmarking Optimization Software with Performance Profiles." *Mathematical Programming* 91 (2): 201–13.

Du, Simon S., Sham M. Kakade, Jason D. Lee, et al. 2021. "Bilinear Classes: A Structural Framework for Provable Generalization in RL." *Proceedings of the 38th International Conference on Machine Learning*.

Du, Simon S., Sham M. Kakade, Ruosong Wang, and Lin F. Yang. 2020. "Is a Good Representation Sufficient for Sample Efficient Reinforcement Learning?" *International Conference on Learning Representations*.

Du, Yilun, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2024. "Improving Factuality and Reasoning in Language Models Through Multiagent Debate." *Proceedings of the 41st International Conference on Machine Learning*.

Duan, Yan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. 2016. "RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning." *arXiv Preprint arXiv:1611.02779*.

Dudík, Miroslav, John Langford, and Lihong Li. 2011. "Doubly Robust Policy Evaluation and Learning." *Proceedings of the 28th International Conference on Machine Learning*.

Dulac-Arnold, Gabriel, Nir Levine, Daniel J. Mankowitz, et al. 2021. "Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks and Analysis." *Machine Learning* 110: 2419–68.

Ecoffet, Adrien, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2021. "First Return, Then Explore." *Nature* 590: 580–86.

Efron, Bradley. 1979. "Bootstrap Methods: Another Look at the Jackknife." *The Annals of Statistics* 7 (1): 1–26.

Efron, Bradley, and Robert J. Tibshirani. 1993. *An Introduction to the Bootstrap*. Chapman; Hall.

Eisenstein, Jacob, Chirag Nagpal, Alekh Agarwal, et al. 2023. "Helping or Herding? Reward Model Ensembles Mitigate but Do Not Eliminate Reward Hacking." *arXiv Preprint*.

Ellis, Benjamin, Jonathan Cook, Skander Moalla, et al. 2023. "SMACv2: An Improved Benchmark for Cooperative Multi-Agent Reinforcement Learning." *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)* 36.

Elman, Jeffrey L. 1990. "Finding Structure in Time." *Cognitive Science* 14 (2): 179–211.

Emmons, Scott, Benjamin Eysenbach, Ilya Kostrikov, and Sergey Levine. 2022. "RvS: What Is Essential for Offline RL via Supervised Learning?" *International Conference on Learning Representations*.

Engstrom, Logan, Andrew Ilyas, Shibani Santurkar, et al. 2020. "Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO." *International Conference on Learning Representations*.

Ernst, Damien, Pierre Geurts, and Louis Wehenkel. 2005. "Tree-Based Batch Mode Reinforcement Learning." *Journal of Machine Learning Research* 6: 503–56.

Espeholt, Lasse, Hubert Soyer, Rémi Munos, et al. 2018. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures." *Proceedings of the 35th International Conference on Machine Learning*, 1407–16.

Ethayarajh, Kawin, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. "KTO: Model Alignment as Prospect Theoretic Optimization." *Proceedings of the 41st International Conference on Machine Learning*.

Even-Dar, Eyal, Shie Mannor, and Yishay Mansour. 2006. "Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems." *Journal of Machine Learning Research* 7: 1079–105.

Even-Dar, Eyal, and Yishay Mansour. 2003. "Learning Rates for Q-Learning." *Journal of Machine Learn-*

ing Research 5: 1–25.

Eysenbach, Benjamin, Abhishek Gupta, Julian Ibarz, and Sergey Levine. 2019. "Diversity Is All You Need: Learning Skills Without a Reward Function." *International Conference on Learning Representations*.

Eysenbach, Benjamin, and Sergey Levine. 2022. "Maximum Entropy RL (Provably) Solves Some Robust RL Problems." *International Conference on Learning Representations*.

Farahmand, Amir-massoud, André Barreto, and Daniel Nikovski. 2017. "Value-Aware Loss Function for Model-Based Reinforcement Learning." *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*.

Fawzi, Alhussein, Matej Balog, Aja Huang, et al. 2022. "Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning." *Nature* 610.

Fazel, Maryam, Rong Ge, Sham Kakade, and Mehran Mesbahi. 2018. "Global Convergence of Policy Gradient Methods for the Linear Quadratic Regulator." *Proceedings of the 35th International Conference on Machine Learning*.

Fearnley, John. 2010. "Exponential Lower Bounds for Policy Iteration." *Automata, Languages and Programming (ICALP 2010)*.

Fedus, William, Prajit Ramachandran, Rishabh Agarwal, et al. 2020. "Revisiting Fundamentals of Experience Replay." *Proceedings of the 37th International Conference on Machine Learning*.

Feinberg, Vladimir, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. 2018. "Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning." *arXiv Preprint arXiv:1803.00101*.

Fink, Arlington M. 1964. "Equilibrium in a Stochastic *n*-Person Game." *Journal of Science of the Hiroshima University, Series A-I* 28: 89–93.

Finn, Chelsea, Pieter Abbeel, and Sergey Levine. 2017. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks." *Proceedings of the 34th International Conference on Machine Learning*.

Finn, Chelsea, Sergey Levine, and Pieter Abbeel. 2016. "Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization." *Proceedings of the 33rd International Conference on Machine Learning*.

Florence, Pete, Corey Lynch, Andy Zeng, et al. 2022. "Implicit Behavioral Cloning." *Proceedings of the 5th Conference on Robot Learning*.

Foerster, Jakob N., Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. 2018. "Counterfactual Multi-Agent Policy Gradients." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Foerster, Jakob, Richard Y. Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. 2018. "Learning with Opponent-Learning Awareness." *Proceedings of the 17th International*

Conference on Autonomous Agents and Multiagent Systems.

Fortunato, Meire, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, et al. 2018. "Noisy Networks for Exploration." *International Conference on Learning Representations*.

Foster, Dean P., and Rakesh V. Vohra. 1997. "Calibrated Learning and Correlated Equilibrium." *Games and Economic Behavior* 21 (1–2): 40–55.

Foster, Dylan J., Sham M. Kakade, Jian Qian, and Alexander Rakhlin. 2021. "The Statistical Complexity of Interactive Decision Making." *arXiv Preprint arXiv:2112.13487*.

Foster, Dylan J., Akshay Krishnamurthy, David Simchi-Levi, and Yunzong Xu. 2022. "Offline Reinforcement Learning: Fundamental Barriers for Value Function Approximation." *Proceedings of the 35th Conference on Learning Theory*.

Foster, Dylan J., and Alexander Rakhlin. 2020. "Beyond UCB: Optimal and Efficient Contextual Bandits with Regression Oracles." *Proceedings of the 37th International Conference on Machine Learning*.

Foster, Dylan J., and Alexander Rakhlin. 2023. "Foundations of Reinforcement Learning and Interactive Decision Making." *arXiv Preprint arXiv:2312.16730*.

Fox, Roy, Ari Pakman, and Naftali Tishby. 2016. "Taming the Noise in Reinforcement Learning via Soft Updates." *Proceedings of the 32nd Conference on Uncertainty in Artificial Intelligence*.

Freeman, C. Daniel, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. "Brax – a Differentiable Physics Engine for Large Scale Rigid Body Simulation." *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.

Fruit, Ronan, and Alessandro Lazaric. 2017. "Exploration–Exploitation in MDPs with Options." *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*.

Fu, Justin, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. 2020. "D4RL: Datasets for Deep Data-Driven Reinforcement Learning." *arXiv Preprint arXiv:2004.07219*.

Fu, Justin, Katie Luo, and Sergey Levine. 2018. "Learning Robust Rewards with Adversarial Inverse Reinforcement Learning." *International Conference on Learning Representations*.

Fu, Justin, Mohammad Norouzi, Ofir Nachum, et al. 2021. "Benchmarks for Deep Off-Policy Evaluation." *International Conference on Learning Representations*.

Fudenberg, Drew, and David K. Levine. 1998. *The Theory of Learning in Games*. MIT Press.

Fujimoto, Scott, Wei-Di Chang, Edward J. Smith, Shixiang Shane Gu, Doina Precup, and David Meger. 2023. "For SALE: State-Action Representation Learning for Deep Reinforcement Learning." *Advances in Neural Information Processing Systems* 36.

Fujimoto, Scott, Edoardo Conti, Mohammad Ghavamzadeh, and Joelle Pineau. 2019. "Benchmarking Batch Deep Reinforcement Learning Algorithms." *arXiv Preprint arXiv:1910.01708*.

Fujimoto, Scott, and Shixiang Shane Gu. 2021. "A Minimalist Approach to Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 34.

Fujimoto, Scott, David Meger, and Doina Precup. 2019. "Off-Policy Deep Reinforcement Learning Without Exploration." *Proceedings of the 36th International Conference on Machine Learning*, 2052–62.

Fujimoto, Scott, Herke van Hoof, and David Meger. 2018. "Addressing Function Approximation Error in Actor-Critic Methods." *Proceedings of the 35th International Conference on Machine Learning*, 1587–96.

Gandhi, Kanishk, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. 2025. "Cognitive Behaviors That Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs." *arXiv Preprint*.

Gao, Leo, John Schulman, and Jacob Hilton. 2023. "Scaling Laws for Reward Model Overoptimization." *Proceedings of the 40th International Conference on Machine Learning*.

García, Javier, and Fernando Fernández. 2015. "A Comprehensive Survey on Safe Reinforcement Learning." *Journal of Machine Learning Research* 16: 1437–80.

Garg, Divyansh, Shuvam Chakraborty, Chris Cundy, Jiaming Song, and Stefano Ermon. 2021. "IQ-Learn: Inverse Soft-Q Learning for Imitation." *Advances in Neural Information Processing Systems* 34.

Garivier, Aurélien, and Olivier Cappé. 2011. "The KL-UCB Algorithm for Bounded Stochastic Bandits and Beyond." *Proceedings of the 24th Annual Conference on Learning Theory*, 359–76.

Garivier, Aurélien, and Emilie Kaufmann. 2016. "Optimal Best Arm Identification with Fixed Confidence." *Proceedings of the 29th Annual Conference on Learning Theory*.

Garivier, Aurélien, and Eric Moulines. 2011. "On Upper-Confidence Bound Policies for Switching Bandit Problems." *Algorithmic Learning Theory (ALT 2011)*.

Geist, Matthieu, and Olivier Pietquin. 2013. "Algorithmic Survey of Parametric Value Function Approximation." *IEEE Transactions on Neural Networks and Learning Systems* 24 (6): 845–67.

Geist, Matthieu, Bruno Scherrer, and Olivier Pietquin. 2019. "A Theory of Regularized Markov Decision Processes." *Proceedings of the 36th International Conference on Machine Learning*.

Ghiassian, Sina, Andrew Patterson, Shivam Garg, Dhawal Gupta, Adam White, and Martha White. 2020. "Gradient Temporal-Difference Learning with Regularized Corrections." *Proceedings of the 37th International Conference on Machine Learning*.

Ghosh, Dibya, Abhishek Gupta, Ashwin Reddy, et al. 2021. "Learning to Reach Goals via Iterated Supervised Learning." *International Conference on Learning Representations*.

Gittins, John C. 1979. "Bandit Processes and Dynamic Allocation Indices." *Journal of the Royal Statistical Society, Series B* 41 (2): 148–77.

Gittins, John, Kevin Glazebrook, and Richard Weber. 2011. *Multi-Armed Bandit Allocation Indices*. 2nd ed. Wiley.

Glorot, Xavier, and Yoshua Bengio. 2010. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*.

Glynn, Peter W. 1990. "Likelihood Ratio Gradient Estimation for Stochastic Systems." *Communications of the ACM* 33 (10): 75–84.

Goldie, Anna, Azalia Mirhoseini, and Jeff Dean. 2024. "That Chip Has Sailed: A Critique of Unfounded Skepticism Around AI for Chip Design." *arXiv Preprint*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, et al. 2014. "Generative Adversarial Nets." *Advances in Neural Information Processing Systems* 27.

Gordon, Geoffrey J. 1995. "Stable Function Approximation in Dynamic Programming." *Proceedings of the Twelfth International Conference on Machine Learning*.

Gordon, Geoffrey J. 2001. "Reinforcement Learning with Function Approximation Converges to a Region." *Advances in Neural Information Processing Systems* 13.

Greensmith, Evan, Peter L. Bartlett, and Jonathan Baxter. 2004. "Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning." *Journal of Machine Learning Research* 5: 1471–530.

Gregor, Karol, Danilo Jimenez Rezende, and Daan Wierstra. 2016. "Variational Intrinsic Control." *arXiv Preprint arXiv:1611.07507*.

Grimm, Christopher, André Barreto, Satinder Singh, and David Silver. 2020. "The Value Equivalence Principle for Model-Based Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Gu, Shixiang, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. 2016. "Continuous Deep Q-Learning with Model-Based Acceleration." *Proceedings of the 33rd International Conference on Machine Learning*.

Guo, Daya, Dejian Yang, Haowei Zhang, Junxiao Song, et al. 2025. "DeepSeek-R1 Incentivizes Reasoning in LLMs Through Reinforcement Learning." *Nature* 645: 633–38.

Ha, David, and Jürgen Schmidhuber. 2018. "World Models." *arXiv Preprint arXiv:1803.10122*.

Haan, Pim de, Dinesh Jayaraman, and Sergey Levine. 2019. "Causal Confusion in Imitation Learning." *Advances in Neural Information Processing Systems* 32.

Haarnoja, Tuomas, Haoran Tang, Pieter Abbeel, and Sergey Levine. 2017. "Reinforcement Learning with Deep Energy-Based Policies." *Proceedings of the 34th International Conference on Machine Learning*.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." *Proceedings of the 35th International Conference on Machine Learning*, 1861–70.

Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, et al. 2018. "Soft Actor-Critic Algorithms and Applications." *arXiv Preprint arXiv:1812.05905*.

Hafner, Danijar, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. 2020. "Dream to Control: Learning Behaviors by Latent Imagination." *International Conference on Learning Representations*.

Hafner, Danijar, Timothy Lillicrap, Ian Fischer, et al. 2019. "Learning Latent Dynamics for Planning from Pixels." *Proceedings of the 36th International Conference on Machine Learning*.

Hafner, Danijar, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. 2021. "Mastering Atari with Discrete World Models." *International Conference on Learning Representations*.

Hafner, Danijar, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. 2025. "Mastering Diverse Control Tasks Through World Models." *Nature* 640: 647–53.

Hallak, Assaf, Dotan Di Castro, and Shie Mannor. 2015. "Contextual Markov Decision Processes." *arXiv Preprint arXiv:1502.02259*.

Hamrick, Jessica B., Abram L. Friesen, Feryal Behbahani, et al. 2021. "On the Role of Planning in Model-Based Deep Reinforcement Learning." *International Conference on Learning Representations*.

Hannan, James. 1957. "Approximation to Bayes Risk in Repeated Play." In *Contributions to the Theory of Games, Volume III*. Annals of Mathematics Studies 39. Princeton University Press.

Hansen, Nicklas, Hao Su, and Xiaolong Wang. 2024. "TD-MPC2: Scalable, Robust World Models for Continuous Control." *International Conference on Learning Representations*.

Hansen, Nicklas, Xiaolong Wang, and Hao Su. 2022. "Temporal Difference Learning for Model Predictive Control." *Proceedings of the 39th International Conference on Machine Learning*.

Hansen, Thomas Dueholm, Peter Bro Miltersen, and Uri Zwick. 2013. "Strategy Iteration Is Strongly Polynomial for 2-Player Turn-Based Stochastic Games with a Constant Discount Factor." *Journal of the ACM* 60 (1).

Harb, Jean, Pierre-Luc Bacon, Martin Klissarov, and Doina Precup. 2018. "When Waiting Is Not an Option: Learning Options with a Deliberation Cost." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael. 1968. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics* 4 (2): 100–107.

Hart, Sergiu, and Andreu Mas-Colell. 2000. "A Simple Adaptive Procedure Leading to Correlated Equilibrium." *Econometrica* 68 (5): 1127–50.

Hart, Sergiu, and Andreu Mas-Colell. 2003. "Uncoupled Dynamics Do Not Lead to Nash Equilibrium." *American Economic Review* 93 (5): 1830–36.

Harutyunyan, Anna, Marc G. Bellemare, Tom Stepleton, and Rémi Munos. 2016. "Q(λ) with Off-Policy Corrections." *Algorithmic Learning Theory*.

Hasselt, Hado van. 2010. "Double Q-Learning." *Advances in Neural Information Processing Systems* 23.

Hasselt, Hado van, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. 2018. "Deep Reinforcement Learning and the Deadly Triad." *arXiv Preprint arXiv:1812.02648*.

Hasselt, Hado van, Arthur Guez, and David Silver. 2016. "Deep Reinforcement Learning with Double Q-Learning." *Proceedings of the AAAI Conference on Artificial Intelligence* 30.

Hasselt, Hado van, Matteo Hessel, and John Aslanides. 2019. "When to Use Parametric Models in Reinforcement Learning?" *Advances in Neural Information Processing Systems* 32.

Hausknecht, Matthew, and Peter Stone. 2015. "Deep Recurrent Q-Learning for Partially Observable MDPs." *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *Proceedings of the IEEE International Conference on Computer Vision*.

Heess, Nicolas, Greg Wayne, David Silver, Timothy Lillicrap, Yuval Tassa, and Tom Erez. 2015. "Learning Continuous Control Policies by Stochastic Value Gradients." *Advances in Neural Information Processing Systems* 28.

Heinrich, Johannes, and David Silver. 2016. "Deep Reinforcement Learning from Self-Play in Imperfect-Information Games." *arXiv Preprint arXiv:1603.01121*.

Henderson, Peter, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. 2020. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning." *Journal of Machine Learning Research* 21.

Henderson, Peter, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. "Deep Reinforcement Learning That Matters." *Proceedings of the AAAI Conference on Artificial Intelligence* 32.

Hessel, Matteo, Joseph Modayil, Hado van Hasselt, et al. 2018. "Rainbow: Combining Improvements in Deep Reinforcement Learning." *Proceedings of the AAAI Conference on Artificial Intelligence* 32.

Hestenes, Magnus R., and Eduard Stiefel. 1952. "Methods of Conjugate Gradients for Solving Linear Systems." *Journal of Research of the National Bureau of Standards* 49 (6): 409–36.

Hiraoka, Takuya, Takahisa Imagawa, Taisei Hashimoto, Takashi Onishi, and Yoshimasa Tsuruoka. 2022. "Dropout Q-Functions for Doubly Efficient Reinforcement Learning." *International Conference on Learning Representations*.

Ho, Jonathan, and Stefano Ermon. 2016. "Generative Adversarial Imitation Learning." *Advances in Neural Information Processing Systems* 29.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9 (8): 1735–80.

Hong, Jiwoo, Noah Lee, and James Thorne. 2024. "ORPO: Monolithic Preference Optimization Without Reference Model." *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*.

Horgan, Dan, John Quan, David Budden, et al. 2018. "Distributed Prioritized Experience Replay." *International Conference on Learning Representations*.

Hornik, Kurt. 1991. "Approximation Capabilities of Multilayer Feedforward Networks." *Neural Networks* 4 (2): 251–57.

Howard, Ronald A. 1960. *Dynamic Programming and Markov Processes*. MIT Press.

Hu, Bin, Kaiqing Zhang, Na Li, Mehran Mesbahi, Maryam Fazel, and Tamer Başar. 2023. "Toward a Theoretical Foundation of Policy Optimization for Learning Control Policies." *Annual Review of Control, Robotics, and Autonomous Systems* 6.

Hu, Junling, and Michael P. Wellman. 2003. "Nash Q-Learning for General-Sum Stochastic Games." *Journal of Machine Learning Research* 4: 1039–69.

Huang, Shengyi, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. 2022. *The 37 Implementation Details of Proximal Policy Optimization*. ICLR Blog Track.

Huang, Shengyi, Rousslan Fernand Julien Dossa, Chang Ye, et al. 2022. "CleanRL: High-Quality Single-File Implementations of Deep Reinforcement Learning Algorithms." *Journal of Machine Learning Research* 23.

Huang, Shengyi, Michael Noukhovitch, Arian Hosseini, Kashif Rasul, Weixun Wang, and Lewis Tunstall. 2024. "The N+ Implementation Details of RLHF with PPO: A Case Study on TL;DR Summarization." *Conference on Language Modeling*.

Huber, Peter J. 1964. "Robust Estimation of a Location Parameter." *Annals of Mathematical Statistics* 35 (1): 73–101.

Hubert, Thomas, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. 2021. "Learning and Planning in Complex Action Spaces." *Proceedings of the 38th International Conference on Machine Learning*.

Hwangbo, Jemin, Joonho Lee, Alexey Dosovitskiy, et al. 2019. "Learning Agile and Dynamic Motor Skills for Legged Robots." *Science Robotics* 4 (26).

Ibarz, Borja, Jan Leike, Tobias Pohlen, Geoffrey Irving, Shane Legg, and Dario Amodei. 2018. "Reward Learning from Human Preferences and Demonstrations in Atari." *Advances in Neural Information*

Processing Systems 31.

Igl, Maximilian, Luisa Zintgraf, Tuan Anh Le, Frank Wood, and Shimon Whiteson. 2018. "Deep Variational Reinforcement Learning for POMDPs." *Proceedings of the 35th International Conference on Machine Learning*.

Ilyas, Andrew, Logan Engstrom, Shibani Santurkar, et al. 2020. "A Closer Look at Deep Policy Gradients." *International Conference on Learning Representations*.

Ioffe, Sergey, and Christian Szegedy. 2015. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *Proceedings of the 32nd International Conference on Machine Learning*.

Irving, Geoffrey, Paul Christiano, and Dario Amodei. 2018. "AI Safety via Debate." *arXiv Preprint arXiv:1805.00899*.

Jaakkola, Tommi, Michael I. Jordan, and Satinder P. Singh. 1994. "On the Convergence of Stochastic Iterative Dynamic Programming Algorithms." *Neural Computation* 6 (6): 1185–201.

Jacob, Athul Paul, David J. Wu, Gabriele Farina, et al. 2022. "Modeling Strong and Human-Like Gameplay with KL-Regularized Search." *Proceedings of the 39th International Conference on Machine Learning*.

Jaderberg, Max et al. 2017. "Population Based Training of Neural Networks." *arXiv Preprint arXiv:1711.09846*.

Jaderberg, Max, Volodymyr Mnih, Wojciech Marian Czarnecki, et al. 2017. "Reinforcement Learning with Unsupervised Auxiliary Tasks." *International Conference on Learning Representations*.

Jaksch, Thomas, Ronald Ortner, and Peter Auer. 2010. "Near-Optimal Regret Bounds for Reinforcement Learning." *Journal of Machine Learning Research* 11: 1563–600.

Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine. 2019. "When to Trust Your Model: Model-Based Policy Optimization." *Advances in Neural Information Processing Systems* 32.

Janner, Michael, Qiyang Li, and Sergey Levine. 2021. "Offline Reinforcement Learning as One Big Sequence Modeling Problem." *Advances in Neural Information Processing Systems* 34.

Jaynes, Edwin T. 1957. "Information Theory and Statistical Mechanics." *Physical Review* 106 (4): 620–30.

Jiang, Nan, Akshay Krishnamurthy, Alekh Agarwal, John Langford, and Robert E. Schapire. 2017. "Contextual Decision Processes with Low Bellman Rank Are PAC-Learnable." *Proceedings of the 34th International Conference on Machine Learning*.

Jiang, Nan, and Lihong Li. 2016. "Doubly Robust Off-Policy Value Evaluation for Reinforcement Learning." *Proceedings of the 33rd International Conference on Machine Learning*.

Jimenez, Carlos E., John Yang, Alexander Wettig, et al. 2024. "SWE-Bench: Can Language Models Re-

solve Real-World GitHub Issues?" *International Conference on Learning Representations*.

Jin, Chi, Zeyuan Allen-Zhu, Sébastien Bubeck, and Michael I. Jordan. 2018. "Is Q-Learning Provably Efficient?" *Advances in Neural Information Processing Systems* 31.

Jin, Chi, Akshay Krishnamurthy, Max Simchowitz, and Tiancheng Yu. 2020. "Reward-Free Exploration for Reinforcement Learning." *Proceedings of the 37th International Conference on Machine Learning*.

Jin, Chi, Qinghua Liu, and Sobhan Miryoosefi. 2021. "Bellman Eluder Dimension: New Rich Classes of RL Problems, and Sample-Efficient Algorithms." *Advances in Neural Information Processing Systems* 34.

Jin, Chi, Zhuoran Yang, Zhaoran Wang, and Michael I. Jordan. 2020. "Provably Efficient Reinforcement Learning with Linear Function Approximation." *Proceedings of the 33rd Conference on Learning Theory*.

Jin, Ying, Zhuoran Yang, and Zhaoran Wang. 2021. "Is Pessimism Provably Efficient for Offline RL?" *Proceedings of the 38th International Conference on Machine Learning*.

Jinnai, Yuu, David Abel, D. Ellis Hershkowitz, Michael Littman, and George Konidaris. 2019. "Finding Options That Minimize Planning Time." *Proceedings of the 36th International Conference on Machine Learning*.

Johnson, Emmeran, Ciara Pike-Burke, and Patrick Rebeschini. 2023. "Optimal Convergence Rate for Exact Policy Mirror Descent in Discounted Markov Decision Processes." *Advances in Neural Information Processing Systems* 36.

Jordan, Scott M., Yash Chandak, Daniel Cohen, Mengxue Zhang, and Philip S. Thomas. 2020. "Evaluating the Performance of Reinforcement Learning Algorithms." *Proceedings of the 37th International Conference on Machine Learning*.

Kaelbling, Leslie Pack. 1993. "Learning to Achieve Goals." *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*.

Kaelbling, Leslie Pack, Michael L. Littman, and Anthony R. Cassandra. 1998. "Planning and Acting in Partially Observable Stochastic Domains." *Artificial Intelligence* 101 (1–2): 99–134.

Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore. 1996. "Reinforcement Learning: A Survey." *Journal of Artificial Intelligence Research* 4: 237–85.

Kaiser, Łukasz, Mohammad Babaeizadeh, Piotr Miłos, et al. 2020. "Model-Based Reinforcement Learning for Atari." *International Conference on Learning Representations*.

Kakade, Sham M. 2001. "A Natural Policy Gradient." *Advances in Neural Information Processing Systems* 14.

Kakade, Sham M. 2003. "On the Sample Complexity of Reinforcement Learning." PhD thesis, University College London.

Kakade, Sham, and John Langford. 2002. "Approximately Optimal Approximate Reinforcement Learning." *Proceedings of the 19th International Conference on Machine Learning*, 267–74.

Kalashnikov, Dmitry, Alex Irpan, Peter Pastor, et al. 2018. "Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation." *Proceedings of the 2nd Conference on Robot Learning*.

Kane, Daniel, Sihan Liu, Shachar Lovett, and Gaurav Mahajan. 2022. "Computational-Statistical Gap in Reinforcement Learning." *Proceedings of the 35th Conference on Learning Theory*.

Kapturowski, Steven, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney. 2019. "Recurrent Experience Replay in Distributed Reinforcement Learning." *International Conference on Learning Representations*.

Kaufmann, Elia, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. 2023. "Champion-Level Drone Racing Using Deep Reinforcement Learning." *Nature* 620 (7976): 982–87.

Kaufmann, Emilie, Nathaniel Korda, and Rémi Munos. 2012. "Thompson Sampling: An Asymptotically Optimal Finite-Time Analysis." *Algorithmic Learning Theory (ALT 2012)*.

Kearns, Michael, Yishay Mansour, and Andrew Y. Ng. 2002. "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes." *Machine Learning* 49 (2–3): 193–208.

Kearns, Michael, and Satinder Singh. 2000. "Bias-Variance Error Bounds for Temporal Difference Updates." *Proceedings of the 13th Annual Conference on Computational Learning Theory*.

Kearns, Michael, and Satinder Singh. 2002. "Near-Optimal Reinforcement Learning in Polynomial Time." *Machine Learning* 49 (2–3): 209–32.

Keshav, Srinivasan. 2007. "How to Read a Paper." *ACM SIGCOMM Computer Communication Review* 37 (3).

Khetarpal, Khimya, Matthew Riemer, Irina Rish, and Doina Precup. 2022. "Towards Continual Reinforcement Learning: A Review and Perspectives." *Journal of Artificial Intelligence Research* 75.

Kidambi, Rahul, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. 2020. "MOReL: Model-Based Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Kim, Moo Jin, Karl Pertsch, Siddharth Karamcheti, et al. 2024. "OpenVLA: An Open-Source Vision-Language-Action Model." *Conference on Robot Learning*.

Kingma, Diederik P., and Jimmy Ba. 2015. "Adam: A Method for Stochastic Optimization." *International Conference on Learning Representations*.

Kingma, Diederik P., and Max Welling. 2014. "Auto-Encoding Variational Bayes." *International Conference on Learning Representations*.

Kirk, Robert, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. 2023. "A Survey of Zero-Shot Generalisation in Deep Reinforcement Learning." *Journal of Artificial Intelligence Research* 76.

Kocsis, Levente, and Csaba Szepesvári. 2006. "Bandit Based Monte-Carlo Planning." *European Conference on Machine Learning*, 282–93.

Koenker, Roger, and Gilbert Bassett. 1978. "Regression Quantiles." *Econometrica* 46 (1): 33–50.

Koller, Daphne, Nimrod Megiddo, and Bernhard von Stengel. 1996. "Efficient Computation of Equilibria for Extensive Two-Person Games." *Games and Economic Behavior* 14 (2): 247–59.

Konda, Vijay R., and John N. Tsitsiklis. 2000. "Actor-Critic Algorithms." *Advances in Neural Information Processing Systems* 12.

Konidaris, George, and Andrew G. Barto. 2009. "Skill Discovery in Continuous Reinforcement Learning Domains Using Skill Chaining." *Advances in Neural Information Processing Systems* 22.

Konidaris, George, Sarah Osentoski, and Philip Thomas. 2011. "Value Function Approximation in Reinforcement Learning Using the Fourier Basis." *Proceedings of the AAAI Conference on Artificial Intelligence* 25.

Kool, Wouter, Herke van Hoof, and Max Welling. 2019. "Buy 4 REINFORCE Samples, Get a Baseline for Free!" *ICLR Workshop on Deep Reinforcement Learning Meets Structured Prediction*.

Korbak, Tomasz, Ethan Perez, and Christopher L. Buckley. 2022. "RL with KL Penalties Is Better Viewed as Bayesian Inference." *Findings of the Association for Computational Linguistics: EMNLP 2022*.

Korf, Richard E. 1990. "Real-Time Heuristic Search." *Artificial Intelligence* 42 (2–3): 189–211.

Kostrikov, Ilya, Ashvin Nair, and Sergey Levine. 2022. "Offline Reinforcement Learning with Implicit Q-Learning." *International Conference on Learning Representations*.

Kostrikov, Ilya, Denis Yarats, and Rob Fergus. 2021. "Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels." *International Conference on Learning Representations*.

Kozuno, Tadashi, Yunhao Tang, Mark Rowland, et al. 2021. "Revisiting Peng's Q(λ) for Modern Reinforcement Learning." *Proceedings of the 38th International Conference on Machine Learning*.

Krakovna, Victoria, Jonathan Uesato, Vladimir Mikulik, et al. 2020. *Specification Gaming: The Flip Side of AI Ingenuity*. DeepMind Blog.

Kuba, Jakub Grudzien, Christian Schroeder de Witt, and Jakob Foerster. 2022. "Mirror Learning: A Unifying Framework of Policy Optimisation." *Proceedings of the 39th International Conference on Machine Learning*.

Kuhn, Harold W. 1950. "A Simplified Two-Person Poker." In *Contributions to the Theory of Games, Volume i*. Annals of Mathematics Studies 24. Princeton University Press.

Kulkarni, Tejas D., Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. 2016. "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation." *Advances in Neural Information Processing Systems* 29.

Kumar, Aviral, Rishabh Agarwal, Dibya Ghosh, and Sergey Levine. 2021. "Implicit Under-Parameterization Inhibits Data-Efficient Deep Reinforcement Learning." *International Conference on Learning Representations*.

Kumar, Aviral, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. 2019. "Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction." *Advances in Neural Information Processing Systems* 32.

Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine. 2020. "Conservative Q-Learning for Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Kurniawati, Hanna, David Hsu, and Wee Sun Lee. 2008. "SARSOP: Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces." *Robotics: Science and Systems*.

Kurutach, Thanard, Ignasi Clavera, Yan Duan, Aviv Tamar, and Pieter Abbeel. 2018. "Model-Ensemble Trust-Region Policy Optimization." *International Conference on Learning Representations*.

Kushner, Harold J., and G. George Yin. 2003. *Stochastic Approximation and Recursive Algorithms and Applications*. 2nd ed. Springer.

Kuznetsov, Arsenii, Pavel Shvechikov, Alexander Grishin, and Dmitry Vetrov. 2020. "Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics." *Proceedings of the 37th International Conference on Machine Learning*.

Lagoudakis, Michail G., and Ronald Parr. 2003. "Least-Squares Policy Iteration." *Journal of Machine Learning Research* 4: 1107–49.

Lai, Tze Leung, and Herbert Robbins. 1985. "Asymptotically Efficient Adaptive Allocation Rules." *Advances in Applied Mathematics* 6 (1): 4–22.

Lakshminarayanan, Balaji, Alexander Pritzel, and Charles Blundell. 2017. "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles." *Advances in Neural Information Processing Systems* 30.

Lambert, Nathan, Brandon Amos, Omry Yadan, and Roberto Calandra. 2020. "Objective Mismatch in Model-Based Reinforcement Learning." *Proceedings of the 2nd Conference on Learning for Dynamics and Control*.

Lambert, Nathan, Jacob Morrison, Valentina Pyatkin, et al. 2024. "Tülu 3: Pushing Frontiers in Open Language Model Post-Training." *arXiv Preprint*.

Lan, Guanghui. 2023. "Policy Mirror Descent for Reinforcement Learning: Linear Convergence, New Sampling Complexity, and Generalized Problem Classes." *Mathematical Programming* 198.

Lan, Qingfeng, Yangchen Pan, Alona Fyshe, and Martha White. 2020. "Maxmin Q-Learning: Controlling the Estimation Bias of Q-Learning." *International Conference on Learning Representations*.

Lanctot, Marc, Kevin Waugh, Martin Zinkevich, and Michael Bowling. 2009. "Monte Carlo Sampling for Regret Minimization in Extensive Games." *Advances in Neural Information Processing Systems* 22.

Lanctot, Marc, Vinicius Zambaldi, Audrunas Gruslys, et al. 2017. "A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning." *Advances in Neural Information Processing Systems* 30.

Lange, Sascha, Thomas Gabel, and Martin Riedmiller. 2012. "Batch Reinforcement Learning." In *Reinforcement Learning: State-of-the-Art*, edited by Marco Wiering and Martijn van Otterlo. Springer.

Laskey, Michael, Jonathan Lee, Roy Fox, Anca Dragan, and Ken Goldberg. 2017. "DART: Noise Injection for Robust Imitation Learning." *Proceedings of the 1st Conference on Robot Learning*.

Laskin, Michael, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. 2020. "Reinforcement Learning with Augmented Data." *Advances in Neural Information Processing Systems* 33.

Laskin, Michael, Luyu Wang, Junhyuk Oh, et al. 2023. "In-Context Reinforcement Learning with Algorithm Distillation." *International Conference on Learning Representations*.

Lattimore, Tor, and Csaba Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press.

Lauer, Martin, and Martin Riedmiller. 2000. "An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems." *Proceedings of the Seventeenth International Conference on Machine Learning*.

Lazaric, Alessandro, Mohammad Ghavamzadeh, and Rémi Munos. 2012. "Finite-Sample Analysis of Least-Squares Policy Iteration." *Journal of Machine Learning Research* 13.

Le, Hoang M., Cameron Voloshin, and Yisong Yue. 2019. "Batch Policy Learning Under Constraints." *Proceedings of the 36th International Conference on Machine Learning*.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–324.

Lee, Harrison, Samrat Phatale, Hassan Mansoor, et al. 2024. "RLAIF Vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback." *Proceedings of the 41st International Conference on Machine Learning*.

Lee, Jonathan N., Annie Xie, Aldo Pacchiano, et al. 2023. "Supervised Pretraining Can Learn in-Context Reinforcement Learning." *Advances in Neural Information Processing Systems* 36.

Lee, Joonho, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. 2020. "Learning Quadrupedal Locomotion over Challenging Terrain." *Science Robotics* 5 (47).

Leonardos, Stefanos, Will Overman, Ioannis Panageas, and Georgios Piliouras. 2022. "Global Conver-

gence of Multi-Agent Policy Gradient in Markov Potential Games." *International Conference on Learning Representations*.

Letcher, Alistair, Jakob Foerster, David Balduzzi, Tim Rocktäschel, and Shimon Whiteson. 2019. "Stable Opponent Shaping in Differentiable Games." *International Conference on Learning Representations*.

Levine, Sergey. 2018. "Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review." *arXiv Preprint arXiv:1805.00909*.

Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu. 2020. "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems." *arXiv Preprint arXiv:2005.01643*.

Levy, Andrew, George Konidaris, Robert Platt, and Kate Saenko. 2019. "Learning Multi-Level Hierarchies with Hindsight." *International Conference on Learning Representations*.

Lewis, Mike, Denis Yarats, Yann Dauphin, Devi Parikh, and Dhruv Batra. 2017. "Deal or No Deal? End-to-End Learning of Negotiation Dialogues." *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*.

Li, Gen, Changxiao Cai, Yuxin Chen, Yuting Wei, and Yuejie Chi. 2024. "Is Q-Learning Minimax Optimal? A Tight Sample Complexity Analysis." *Operations Research*.

Li, Gen, Yuting Wei, Yuejie Chi, and Yuxin Chen. 2023. "Softmax Policy Gradient Methods Can Take Exponential Time to Converge." *Mathematical Programming*.

Li, Gen, Yuting Wei, Yuejie Chi, Yuantao Gu, and Yuxin Chen. 2020a. "Breaking the Sample Size Barrier in Model-Based Reinforcement Learning with a Generative Model." *Advances in Neural Information Processing Systems* 33.

Li, Gen, Yuting Wei, Yuejie Chi, Yuantao Gu, and Yuxin Chen. 2020b. "Sample Complexity of Asynchronous Q-Learning: Sharper Analysis and Variance Reduction." *Advances in Neural Information Processing Systems* 33.

Li, Lihong, Wei Chu, John Langford, and Robert E. Schapire. 2010. "A Contextual-Bandit Approach to Personalized News Article Recommendation." *Proceedings of the 19th International Conference on World Wide Web*, 661–70.

Li, Lihong, Wei Chu, John Langford, and Xuanhui Wang. 2011. "Unbiased Offline Evaluation of Contextual-Bandit-Based News Article Recommendation Algorithms." *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*.

Lightman, Hunter, Vineet Kosaraju, Yura Burda, et al. 2024. "Let's Verify Step by Step." *International Conference on Learning Representations*.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, et al. 2016. "Continuous Control with Deep Reinforcement Learning." *International Conference on Learning Representations*.

Lin, Long-Ji. 1992. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and

Teaching." *Machine Learning* 8 (3–4): 293–321.

Littman, Michael L. 1994. "Markov Games as a Framework for Multi-Agent Reinforcement Learning." *Proceedings of the Eleventh International Conference on Machine Learning*, 157–63.

Littman, Michael L. 2001. "Friend-or-Foe Q-Learning in General-Sum Games." *Proceedings of the Eighteenth International Conference on Machine Learning*.

Littman, Michael L., Thomas L. Dean, and Leslie Pack Kaelbling. 1995. "On the Complexity of Solving Markov Decision Problems." *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*.

Littman, Michael L., Richard S. Sutton, and Satinder Singh. 2001. "Predictive Representations of State." *Advances in Neural Information Processing Systems* 14.

Littman, Michael L., and Csaba Szepesvári. 1996. "A Generalized Reinforcement-Learning Model: Convergence and Applications." *Proceedings of the Thirteenth International Conference on Machine Learning*.

Liu, Boyi, Qi Cai, Zhuoran Yang, and Zhaoran Wang. 2019. "Neural Trust Region/Proximal Policy Optimization Attains Globally Optimal Policy." *Advances in Neural Information Processing Systems* 32.

Liu, Mingjie, Shizhe Diao, Ximing Lu, et al. 2025. "ProRL: Prolonged Reinforcement Learning Expands Reasoning Boundaries in Large Language Models." *arXiv Preprint*.

Liu, Qiang, Lihong Li, Ziyang Tang, and Dengyong Zhou. 2018. "Breaking the Curse of Horizon: Infinite-Horizon Off-Policy Estimation." *Advances in Neural Information Processing Systems* 31.

Liu, Zichen, Changyu Chen, Wenjun Li, et al. 2025. "Understanding R1-Zero-Like Training: A Critical Perspective." *arXiv Preprint arXiv:2503.20783*.

Loshchilov, Ilya, and Frank Hutter. 2019. "Decoupled Weight Decay Regularization." *International Conference on Learning Representations*.

Lowe, Ryan, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments." *Advances in Neural Information Processing Systems* 30.

Lyle, Clare, Pablo Samuel Castro, and Marc G. Bellemare. 2019. "A Comparative Analysis of Expected and Distributional Reinforcement Learning." *Proceedings of the AAAI Conference on Artificial Intelligence* 33.

Lyle, Clare, Mark Rowland, and Will Dabney. 2022. "Understanding and Preventing Capacity Loss in Reinforcement Learning." *International Conference on Learning Representations*.

Lyle, Clare, Zeyu Zheng, Evgenii Nikishin, Bernardo Avila Pires, Razvan Pascanu, and Will Dabney. 2023. "Understanding Plasticity in Neural Networks." *Proceedings of the 40th International Conference on Machine Learning*.

Machado, Marlos C., André Barreto, Doina Precup, and Michael Bowling. 2023. "Temporal Abstraction in Reinforcement Learning with the Successor Representation." *Journal of Machine Learning Research* 24.

Machado, Marlos C., Marc G. Bellemare, and Michael Bowling. 2017. "A Laplacian Framework for Option Discovery in Reinforcement Learning." *Proceedings of the 34th International Conference on Machine Learning*.

Machado, Marlos C., Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. 2018. "Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents." *Journal of Artificial Intelligence Research* 61.

Madani, Omid, Steve Hanks, and Anne Condon. 2003. "On the Undecidability of Probabilistic Planning and Related Stochastic Optimization Problems." *Artificial Intelligence* 147 (1–2): 5–34.

Maei, Hamid Reza. 2011. "Gradient Temporal-Difference Learning Algorithms." PhD thesis, University of Alberta.

Mahadevan, Sridhar. 1996. "Average Reward Reinforcement Learning: Foundations, Algorithms, and Empirical Results." *Machine Learning* 22: 159–95.

Mahadevan, Sridhar, and Mauro Maggioni. 2007. "Proto-Value Functions: A Laplacian Framework for Learning Representation and Control in Markov Decision Processes." *Journal of Machine Learning Research* 8.

Mahmood, A. Rupam, Hado van Hasselt, and Richard S. Sutton. 2014. "Weighted Importance Sampling for Off-Policy Learning with Linear Function Approximation." *Advances in Neural Information Processing Systems* 27.

Makoviychuk, Viktor et al. 2021. "Isaac Gym: High Performance GPU-Based Physics Simulation for Robot Learning." *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.

Manheim, David, and Scott Garrabrant. 2018. "Categorizing Variants of Goodhart's Law." *arXiv Preprint*.

Mankowitz, Daniel J., Andrea Michi, Anton Zhernov, et al. 2023. "Faster Sorting Algorithms Discovered Using Deep Reinforcement Learning." *Nature* 618.

Mann, Timothy A., and Shie Mannor. 2014. "Scaling up Approximate Value Iteration with Options: Better Policies with Fewer Iterations." *Proceedings of the 31st International Conference on Machine Learning*.

Manne, Alan S. 1960. "Linear Programming and Sequential Decisions." *Management Science* 6 (3): 259–67.

Marbach, Peter, and John N. Tsitsiklis. 2001. "Simulation-Based Optimization of Markov Reward Processes." *IEEE Transactions on Automatic Control* 46 (2): 191–209.

Martens, James, and Roger Grosse. 2015. "Optimizing Neural Networks with Kronecker-Factored Ap-

proximate Curvature." *Proceedings of the 32nd International Conference on Machine Learning*.

Matignon, Laetitia, Guillaume J. Laurent, and Nadine Le Fort-Piat. 2012. "Independent Reinforcement Learners in Cooperative Markov Games: A Survey Regarding Coordination Problems." *The Knowledge Engineering Review* 27 (1): 1–31.

Maurer, Andreas, and Massimiliano Pontil. 2009. "Empirical Bernstein Bounds and Sample Variance Penalization." *Proceedings of the 22nd Conference on Learning Theory*.

McGovern, Amy, and Andrew G. Barto. 2001. "Automatic Discovery of Subgoals in Reinforcement Learning Using Diverse Density." *Proceedings of the Eighteenth International Conference on Machine Learning*.

McMahan, H. Brendan, Geoffrey J. Gordon, and Avrim Blum. 2003. "Planning in the Presence of Cost Functions Controlled by an Adversary." *Proceedings of the Twentieth International Conference on Machine Learning*.

Mei, Jincheng, Yue Gao, Bo Dai, Csaba Szepesvári, and Dale Schuurmans. 2021. "Leveraging Non-Uniformity in First-Order Non-Convex Optimization." *Proceedings of the 38th International Conference on Machine Learning*.

Mei, Jincheng, Chenjun Xiao, Csaba Szepesvári, and Dale Schuurmans. 2020. "On the Global Convergence Rates of Softmax Policy Gradient Methods." *Proceedings of the 37th International Conference on Machine Learning*.

Ménard, Pierre, and Aurélien Garivier. 2017. "A Minimax and Asymptotically Optimal Algorithm for Stochastic Bandits." *Algorithmic Learning Theory (ALT 2017)*.

Meng, Yu, Mengzhou Xia, and Danqi Chen. 2024. "SimPO: Simple Preference Optimization with a Reference-Free Reward." *Advances in Neural Information Processing Systems* 37.

Mertikopoulos, Panayotis, Christos Papadimitriou, and Georgios Piliouras. 2018. "Cycles in Adversarial Regularized Learning." *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*.

Meta Fundamental AI Research Diplomacy Team (FAIR), Anton Bakhtin, Noam Brown, et al. 2022. "Human-Level Play in the Game of Diplomacy by Combining Language Models with Strategic Reasoning." *Science* 378 (6624): 1067–74.

Metropolis, Nicholas, and Stanislaw Ulam. 1949. "The Monte Carlo Method." *Journal of the American Statistical Association* 44 (247): 335–41.

Minsky, Marvin. 1961. "Steps Toward Artificial Intelligence." *Proceedings of the IRE* 49 (1): 8–30.

Mirhoseini, Azalia, Anna Goldie, Mustafa Yazgan, et al. 2021. "A Graph Placement Methodology for Fast Chip Design." *Nature* 594.

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, et al. 2016. "Asynchronous Methods for

Deep Reinforcement Learning." *Proceedings of the 33rd International Conference on Machine Learning*, 1928–37.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, et al. 2013. "Playing Atari with Deep Reinforcement Learning." *arXiv Preprint arXiv:1312.5602*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33.

Moerland, Thomas M., Joost Broekens, Aske Plaat, and Catholijn M. Jonker. 2023. "Model-Based Reinforcement Learning: A Survey." *Foundations and Trends in Machine Learning* 16 (1): 1–118.

Mohamed, Shakir, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. 2020. "Monte Carlo Gradient Estimation in Machine Learning." *Journal of Machine Learning Research* 21 (132): 1–62.

Montague, P. Read, Peter Dayan, and Terrence J. Sejnowski. 1996. "A Framework for Mesencephalic Dopamine Systems Based on Predictive Hebbian Learning." *Journal of Neuroscience* 16 (5): 1936–47.

Moore, Andrew W. 1990. "Efficient Memory-Based Learning for Robot Control." PhD thesis, University of Cambridge.

Moore, Andrew W., and Christopher G. Atkeson. 1993. "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time." *Machine Learning* 13 (1): 103–30.

Morad, Steven, Ryan Kortvelesy, Matteo Bettini, Stephan Liwicki, and Amanda Prorok. 2023. "POPGym: Benchmarking Partially Observable Reinforcement Learning." *International Conference on Learning Representations*.

Moravčík, Matej, Martin Schmid, Neil Burch, et al. 2017. "DeepStack: Expert-Level Artificial Intelligence in Heads-up No-Limit Poker." *Science* 356 (6337): 508–13.

Munos, Rémi. 2003. "Error Bounds for Approximate Policy Iteration." *Proceedings of the 20th International Conference on Machine Learning*.

Munos, Rémi. 2007. "Performance Bounds in L_p-Norm for Approximate Value Iteration." *SIAM Journal on Control and Optimization* 46 (2): 541–61.

Munos, Rémi, Tom Stepleton, Anna Harutyunyan, and Marc G. Bellemare. 2016. "Safe and Efficient Off-Policy Reinforcement Learning." *Advances in Neural Information Processing Systems* 29.

Munos, Rémi, and Csaba Szepesvári. 2008. "Finite-Time Bounds for Fitted Value Iteration." *Journal of Machine Learning Research* 9: 815–57.

Munos, Rémi, Michal Valko, Daniele Calandriello, et al. 2024. "Nash Learning from Human Feedback." *Proceedings of the 41st International Conference on Machine Learning*.

Nachum, Ofir, Yinlam Chow, Bo Dai, and Lihong Li. 2019. "DualDICE: Behavior-Agnostic Estimation of Discounted Stationary Distribution Corrections." *Advances in Neural Information Processing Systems*

32.

Nachum, Ofir, Shixiang Gu, Honglak Lee, and Sergey Levine. 2018. "Data-Efficient Hierarchical Reinforcement Learning." *Advances in Neural Information Processing Systems* 31.

Nachum, Ofir, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. 2017. "Bridging the Gap Between Value and Policy Based Reinforcement Learning." *Advances in Neural Information Processing Systems* 30.

Nachum, Ofir, Haoran Tang, Xingyu Lu, Shixiang Gu, Honglak Lee, and Sergey Levine. 2019. "Why Does Hierarchy (Sometimes) Work so Well in Reinforcement Learning?" *arXiv Preprint arXiv:1909.10618*.

Nagabandi, Anusha, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. 2018. "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning." *IEEE International Conference on Robotics and Automation*.

Nair, Ashvin, Abhishek Gupta, Murtaza Dalal, and Sergey Levine. 2020. "AWAC: Accelerating Online Reinforcement Learning with Offline Datasets." *arXiv Preprint arXiv:2006.09359*.

Nakamoto, Mitsuhiko, Yuexiang Zhai, Anikait Singh, et al. 2023. "Cal-QL: Calibrated Offline RL Pre-Training for Efficient Online Fine-Tuning." *Advances in Neural Information Processing Systems* 36.

Nash, John. 1951. "Non-Cooperative Games." *Annals of Mathematics* 54 (2): 286–95.

Nauman, Michal, Mateusz Ostaszewski, Krzysztof Jankowski, Piotr Miłoś, and Marek Cygan. 2024. "Bigger, Regularized, Optimistic: Scaling for Compute and Sample-Efficient Continuous Control." *Advances in Neural Information Processing Systems* 37.

Neumann, John von. 1928. "Zur Theorie Der Gesellschaftsspiele." *Mathematische Annalen* 100: 295–320.

Ng, Andrew Y., Daishi Harada, and Stuart Russell. 1999. "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping." *Proceedings of the 16th International Conference on Machine Learning*, 278–87.

Ng, Andrew Y., and Stuart Russell. 2000. "Algorithms for Inverse Reinforcement Learning." *Proceedings of the 17th International Conference on Machine Learning*, 663–70.

Ni, Tianwei, Benjamin Eysenbach, and Ruslan Salakhutdinov. 2022. "Recurrent Model-Free RL Can Be a Strong Baseline for Many POMDPs." *Proceedings of the 39th International Conference on Machine Learning*.

Ni, Tianwei, Michel Ma, Benjamin Eysenbach, and Pierre-Luc Bacon. 2023. "When Do Transformers Shine in RL? Decoupling Memory from Credit Assignment." *Advances in Neural Information Processing Systems* 36.

Nie, Xinkun, Xiaoying Tian, Jonathan Taylor, and James Zou. 2018. "Why Adaptively Collected Data Have Negative Bias and How to Correct for It." *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*.

Nikishin, Evgenii, Max Schwarzer, Pierluca D'Oro, Pierre-Luc Bacon, and Aaron Courville. 2022. "The Primacy Bias in Deep Reinforcement Learning." *Proceedings of the 39th International Conference on Machine Learning*.

Nisan, Noam, Tim Roughgarden, Éva Tardos, and Vijay V. Vazirani, eds. 2007. *Algorithmic Game Theory*. Cambridge University Press.

Niv, Yael. 2009. "Reinforcement Learning in the Brain." *Journal of Mathematical Psychology* 53 (3): 139–54.

Nota, Chris, and Philip S. Thomas. 2020. "Is the Policy Gradient a Gradient?" *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*.

O'Donoghue, Brendan, Rémi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. 2017. "Combining Policy Gradient and Q-Learning." *International Conference on Learning Representations*.

Obando-Ceron, Johan S., and Pablo Samuel Castro. 2021. "Revisiting Rainbow: Promoting More Insightful and Inclusive Deep Reinforcement Learning Research." *Proceedings of the 38th International Conference on Machine Learning*.

Oh, Junhyuk, Greg Farquhar, Iurii Kemaev, et al. 2025. "Discovering State-of-the-Art Reinforcement Learning Algorithms." *Nature*.

Oh, Junhyuk, Matteo Hessel, Wojciech M. Czarnecki, et al. 2020. "Discovering Reinforcement Learning Algorithms." *Advances in Neural Information Processing Systems* 33.

Oh, Junhyuk, Satinder Singh, and Honglak Lee. 2017. "Value Prediction Network." *Advances in Neural Information Processing Systems* 30.

Oliehoek, Frans A., and Christopher Amato. 2016. *A Concise Introduction to Decentralized POMDPs*. Springer.

Open Ended Learning Team, Adam Stooke, Anuj Mahajan, et al. 2021. "Open-Ended Learning Leads to Generally Capable Agents." *arXiv Preprint arXiv:2107.12808*.

OpenAI. 2024. *Learning to Reason with LLMs*. Blog post.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, et al. 2019. "Solving Rubik's Cube with a Robot Hand." *arXiv Preprint arXiv:1910.07113*.

Osa, Takayuki, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. 2018. "An Algorithmic Perspective on Imitation Learning." *Foundations and Trends in Robotics* 7 (1–2).

Osband, Ian, John Aslanides, and Albin Cassirer. 2018. "Randomized Prior Functions for Deep Reinforcement Learning." *Advances in Neural Information Processing Systems* 31.

Osband, Ian, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. "Deep Exploration via Bootstrapped DQN." *Advances in Neural Information Processing Systems* 29.

Osband, Ian, Yotam Doron, Matteo Hessel, et al. 2020. "Behaviour Suite for Reinforcement Learning." *International Conference on Learning Representations*.

Osband, Ian, Daniel Russo, and Benjamin Van Roy. 2013. "(More) Efficient Reinforcement Learning via Posterior Sampling." *Advances in Neural Information Processing Systems* 26.

Osband, Ian, Benjamin Van Roy, Daniel J. Russo, and Zheng Wen. 2019. "Deep Exploration via Randomized Value Functions." *Journal of Machine Learning Research* 20 (124): 1–62.

Osband, Ian, Benjamin Van Roy, and Zheng Wen. 2016. "Generalization and Exploration via Randomized Value Functions." *Proceedings of the 33rd International Conference on Machine Learning*.

Ostrovski, Georg, Marc G. Bellemare, Aäron van den Oord, and Rémi Munos. 2017. "Count-Based Exploration with Neural Density Models." *Proceedings of the 34th International Conference on Machine Learning*.

Ouyang, Long, Jeffrey Wu, Xu Jiang, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." *Advances in Neural Information Processing Systems* 35.

Owen, Art B. 2013. *Monte Carlo Theory, Methods and Examples*.

Paine, Tom Le, Cosmin Paduraru, Andrea Michi, et al. 2020. "Hyperparameter Selection for Offline Reinforcement Learning." *arXiv Preprint arXiv:2007.09055*.

Papadimitriou, Christos H. 1994. "On the Complexity of the Parity Argument and Other Inefficient Proofs of Existence." *Journal of Computer and System Sciences* 48 (3): 498–532.

Papadimitriou, Christos H., and John N. Tsitsiklis. 1987. "The Complexity of Markov Decision Processes." *Mathematics of Operations Research* 12 (3): 441–50.

Pardo, Fabio, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. 2018. "Time Limits in Reinforcement Learning." *Proceedings of the 35th International Conference on Machine Learning*.

Parisotto, Emilio, H. Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre, et al. 2020. "Stabilizing Transformers for Reinforcement Learning." *Proceedings of the 37th International Conference on Machine Learning*.

Park, Joon Sung, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. "Generative Agents: Interactive Simulacra of Human Behavior." *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*.

Parr, Ronald, and Stuart Russell. 1998. "Reinforcement Learning with Hierarchies of Machines." *Advances in Neural Information Processing Systems* 10.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." *Proceedings of the 30th International Conference on Machine Learning*.

Paster, Keiran, Sheila McIlraith, and Jimmy Ba. 2022. "You Can't Count on Luck: Why Decision Trans-

formers and RvS Fail in Stochastic Environments." *Advances in Neural Information Processing Systems* 35.

Paszke, Adam et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems* 32.

Pateria, Shubham, Budhitama Subagdja, Ah-Hwee Tan, and Chai Quek. 2021. "Hierarchical Reinforcement Learning: A Comprehensive Survey." *ACM Computing Surveys* 54 (5).

Paternain, Santiago, Luiz F. O. Chamon, Miguel Calvo-Fullana, and Alejandro Ribeiro. 2019. "Constrained Reinforcement Learning Has Zero Duality Gap." *Advances in Neural Information Processing Systems* 32.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. 2017. "Curiosity-Driven Exploration by Self-Supervised Prediction." *Proceedings of the 34th International Conference on Machine Learning*.

Patterson, Andrew, Samuel Neumann, Martha White, and Adam White. 2024. "Empirical Design in Reinforcement Learning." *Journal of Machine Learning Research* 25.

Pearlmutter, Barak A. 1994. "Fast Exact Multiplication by the Hessian." *Neural Computation* 6 (1): 147–60.

Peng, Jing, and Ronald J. Williams. 1993. "Efficient Learning and Planning Within the Dyna Framework." *Adaptive Behavior* 1 (4).

Peng, Jing, and Ronald J. Williams. 1996. "Incremental Multi-Step Q-Learning." *Machine Learning* 22 (1–3): 283–90.

Peng, Xue Bin, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. 2018. "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization." *IEEE International Conference on Robotics and Automation*.

Peng, Xue Bin, Aviral Kumar, Grace Zhang, and Sergey Levine. 2019. "Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning." *arXiv Preprint arXiv:1910.00177*.

Perkins, Theodore J., and Doina Precup. 2003. "A Convergent Form of Approximate Policy Iteration." *Advances in Neural Information Processing Systems* 15.

Perolat, Julien, Bart De Vylder, Daniel Hennes, et al. 2022. "Mastering the Game of Stratego with Model-Free Multiagent Reinforcement Learning." *Science* 378 (6623): 990–96.

Perolat, Julien, Remi Munos, Jean-Baptiste Lespiau, et al. 2021. "From Poincaré Recurrence to Convergence in Imperfect Information Games: Finding Equilibrium via Regularization." *Proceedings of the 38th International Conference on Machine Learning*.

Peters, Jan, Katharina Mülling, and Yasemin Altün. 2010. "Relative Entropy Policy Search." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Peters, Jan, and Stefan Schaal. 2008a. "Natural Actor-Critic." *Neurocomputing* 71 (7–9): 1180–90.

Peters, Jan, and Stefan Schaal. 2008b. "Reinforcement Learning of Motor Skills with Policy Gradients." *Neural Networks* 21 (4): 682–97.

Pineau, Joelle, Geoff Gordon, and Sebastian Thrun. 2003. "Point-Based Value Iteration: An Anytime Algorithm for POMDPs." *Proceedings of the 18th International Joint Conference on Artificial Intelligence*.

Pineau, Joelle, Philippe Vincent-Lamarre, Koustuv Sinha, et al. 2021. "Improving Reproducibility in Machine Learning Research (a Report from the NeurIPS 2019 Reproducibility Program)." *Journal of Machine Learning Research* 22.

Plappert, Matthias, Rein Houthooft, Prafulla Dhariwal, et al. 2018. "Parameter Space Noise for Exploration." *International Conference on Learning Representations*.

Polyak, Boris T. 1964. "Some Methods of Speeding up the Convergence of Iteration Methods." *USSR Computational Mathematics and Mathematical Physics* 4 (5): 1–17.

Polyak, Boris T., and Anatoli B. Juditsky. 1992. "Acceleration of Stochastic Approximation by Averaging." *SIAM Journal on Control and Optimization* 30 (4): 838–55.

Pomerleau, Dean A. 1989. "ALVINN: An Autonomous Land Vehicle in a Neural Network." *Advances in Neural Information Processing Systems* 1.

Precup, Doina. 2000. "Temporal Abstraction in Reinforcement Learning." PhD thesis, University of Massachusetts Amherst.

Precup, Doina, Richard S. Sutton, and Satinder Singh. 2000. "Eligibility Traces for Off-Policy Policy Evaluation." *Proceedings of the 17th International Conference on Machine Learning*, 759–66.

Puterman, Martin L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley.

Puterman, Martin L., and Moon Chirl Shin. 1978. "Modified Policy Iteration Algorithms for Discounted Markov Decision Problems." *Management Science* 24 (11): 1127–37.

Rafailov, Rafael, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model." *Advances in Neural Information Processing Systems* 36.

Raffin, Antonin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. "Stable-Baselines3: Reliable Reinforcement Learning Implementations." *Journal of Machine Learning Research* 22.

Rakelly, Kate, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. 2019. "Efficient Off-Policy Meta-Reinforcement Learning via Probabilistic Context Variables." *Proceedings of the 36th International Conference on Machine Learning*.

Randløv, Jette, and Preben Alstrøm. 1998. "Learning to Drive a Bicycle Using Reinforcement Learning

and Shaping." *Proceedings of the 15th International Conference on Machine Learning*.

Ranzato, Marc'Aurelio, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. "Sequence Level Training with Recurrent Neural Networks." *International Conference on Learning Representations*.

Rashid, Tabish, Gregory Farquhar, Bei Peng, and Shimon Whiteson. 2020. "Weighted QMIX: Expanding Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Rashid, Tabish, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2018. "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning." *Proceedings of the 35th International Conference on Machine Learning*.

Rashidinejad, Paria, Banghua Zhu, Cong Ma, Jiantao Jiao, and Stuart Russell. 2021. "Bridging Offline Reinforcement Learning and Imitation Learning: A Tale of Pessimism." *Advances in Neural Information Processing Systems* 34.

Ratliff, Nathan D., J. Andrew Bagnell, and Martin A. Zinkevich. 2006. "Maximum Margin Planning." *Proceedings of the 23rd International Conference on Machine Learning*.

Ray, Alex, Joshua Achiam, and Dario Amodei. 2019. *Benchmarking Safe Exploration in Deep Reinforcement Learning*. OpenAI.

Razin, Noam, Sadhika Malladi, Adithya Bhaskar, Danqi Chen, Sanjeev Arora, and Boris Hanin. 2025. "Unintentional Unalignment: Likelihood Displacement in Direct Preference Optimization." *International Conference on Learning Representations*.

Reddy, Siddharth, Anca D. Dragan, and Sergey Levine. 2020. "SQIL: Imitation Learning via Reinforcement Learning with Sparse Rewards." *International Conference on Learning Representations*.

Reed, Scott, Konrad Zolna, Emilio Parisotto, et al. 2022. "A Generalist Agent." *Transactions on Machine Learning Research*.

Rescorla, Robert A., and Allan R. Wagner. 1972. "A Theory of Pavlovian Conditioning: Variations in the Effectiveness of Reinforcement and Nonreinforcement." In *Classical Conditioning II: Current Research and Theory*, edited by Abraham H. Black and William F. Prokasy. Appleton-Century-Crofts.

Riedmiller, Martin. 2005. "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method." *Machine Learning: ECML 2005*, Lecture notes in computer science, vol. 3720.

Robbins, Herbert. 1952. "Some Aspects of the Sequential Design of Experiments." *Bulletin of the American Mathematical Society* 58 (5): 527–35.

Robbins, Herbert, and Sutton Monro. 1951. "A Stochastic Approximation Method." *The Annals of Mathematical Statistics* 22 (3): 400–407.

Robbins, Herbert, and David Siegmund. 1971. "A Convergence Theorem for Non Negative Almost Super-

martingales and Some Applications." In *Optimizing Methods in Statistics*, edited by Jagdish S. Rustagi. Academic Press.

Robinson, Julia. 1951. "An Iterative Method of Solving a Game." *Annals of Mathematics* 54 (2): 296–301.

Rockafellar, R. Tyrrell, and Stanislav Uryasev. 2000. "Optimization of Conditional Value-at-Risk." *Journal of Risk* 2 (3).

Romera-Paredes, Bernardino, Mohammadamin Barekatain, Alexander Novikov, et al. 2024. "Mathematical Discoveries from Program Search with Large Language Models." *Nature* 625.

Ross, Stéphane, and J. Andrew Bagnell. 2010. "Efficient Reductions for Imitation Learning." *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*.

Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell. 2011. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 627–35.

Rowland, Mark, Marc G. Bellemare, Will Dabney, Rémi Munos, and Yee Whye Teh. 2018. "An Analysis of Categorical Distributional Reinforcement Learning." *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*.

Rowland, Mark, Will Dabney, and Rémi Munos. 2020. "Adaptive Trade-Offs in Off-Policy Learning." *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics*.

Rudin, Nikita, David Hoeller, Philipp Reist, and Marco Hutter. 2022. "Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning." *Proceedings of the 5th Conference on Robot Learning*.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323: 533–36.

Rummery, Gavin A., and Mahesan Niranjan. 1994. *On-Line Q-Learning Using Connectionist Systems*. CUED/F-INFENG/TR 166. Cambridge University Engineering Department.

Russell, Stuart, and Peter Norvig. 2021. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson.

Russo, Daniel J., Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. "A Tutorial on Thompson Sampling." *Foundations and Trends in Machine Learning* 11 (1): 1–96.

Russo, Daniel, and Benjamin Van Roy. 2013. "Eluder Dimension and the Sample Complexity of Optimistic Exploration." *Advances in Neural Information Processing Systems* 26.

Russo, Daniel, and Benjamin Van Roy. 2014. "Learning to Optimize via Posterior Sampling." *Mathematics of Operations Research* 39 (4): 1221–43.

Samuel, Arthur L. 1959. "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development* 3 (3): 210–29.

Samvelyan, Mikayel, Tabish Rashid, Christian Schroeder de Witt, et al. 2019. "The StarCraft Multi-Agent Challenge." *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems*.

Saxe, Andrew M., James L. McClelland, and Surya Ganguli. 2014. "Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks." *International Conference on Learning Representations*.

Schaul, Tom, Daniel Horgan, Karol Gregor, and David Silver. 2015. "Universal Value Function Approximators." *Proceedings of the 32nd International Conference on Machine Learning*.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 2016. "Prioritized Experience Replay." *International Conference on Learning Representations*.

Scherrer, Bruno. 2010. "Should One Compute the Temporal Difference Fix Point or Minimize the Bellman Residual? The Unified Oblique Projection View." *Proceedings of the 27th International Conference on Machine Learning*.

Scherrer, Bruno. 2016. "Improved and Generalized Upper Bounds on the Complexity of Policy Iteration." *Mathematics of Operations Research* 41.

Schmidhuber, Jürgen. 1987. *Evolutionary Principles in Self-Referential Learning*. Diploma thesis, Technische Universität München.

Schmidhuber, Jürgen. 2010. "Formal Theory of Creativity, Fun, and Intrinsic Motivation (1990–2010)." *IEEE Transactions on Autonomous Mental Development* 2 (3): 230–47.

Schramm, Liam, Yunfu Deng, Edgar Granados, and Abdeslam Boularias. 2022. "USHER: Unbiased Sampling for Hindsight Experience Replay." *Conference on Robot Learning*.

Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, et al. 2020. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model." *Nature* 588 (7839): 604–9.

Schroeder de Witt, Christian, Tarun Gupta, Denys Makoviichuk, et al. 2020. "Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge?" *arXiv Preprint arXiv:2011.09533*.

Schulman, John. 2020. *Approximating KL Divergence*. Blog post, http://joschu.net/blog/kl-approx.html.

Schulman, John, Xi Chen, and Pieter Abbeel. 2017. "Equivalence Between Policy Gradients and Soft Q-Learning." *arXiv Preprint arXiv:1704.06440*.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. "Trust Region Policy Optimization." *Proceedings of the 32nd International Conference on Machine Learning*, 1889–97.

Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2016. "High-Dimensional Continuous Control Using Generalized Advantage Estimation." *International Conference on Learning Representations*.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. "Proximal Policy Optimization Algorithms." *arXiv Preprint arXiv:1707.06347*.

Schultz, Wolfram, Peter Dayan, and P. Read Montague. 1997. "A Neural Substrate of Prediction and Reward." *Science* 275 (5306): 1593–99.

Schwarzer, Max, Ankesh Anand, Rishab Goel, R. Devon Hjelm, Aaron Courville, and Philip Bachman. 2021. "Data-Efficient Reinforcement Learning with Self-Predictive Representations." *International Conference on Learning Representations*.

Schwarzer, Max, Johan Obando-Ceron, Aaron Courville, Marc G. Bellemare, Rishabh Agarwal, and Pablo Samuel Castro. 2023. "Bigger, Better, Faster: Human-Level Atari with Human-Level Efficiency." *Proceedings of the 40th International Conference on Machine Learning*.

Seijen, Harm van, A. Rupam Mahmood, Patrick M. Pilarski, Marlos C. Machado, and Richard S. Sutton. 2016. "True Online Temporal-Difference Learning." *Journal of Machine Learning Research* 17 (145): 1–40.

Seijen, Harm van, and Richard S. Sutton. 2014. "True Online TD(λ)." *Proceedings of the 31st International Conference on Machine Learning*, 692–700.

Seijen, Harm van, Hado van Hasselt, Shimon Whiteson, and Marco Wiering. 2009. "A Theoretical and Empirical Analysis of Expected Sarsa." *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 177–84.

Shani, Guy, Joelle Pineau, and Robert Kaplow. 2013. "A Survey of Point-Based POMDP Solvers." *Autonomous Agents and Multi-Agent Systems* 27 (1): 1–51.

Shani, Lior, Yonathan Efroni, and Shie Mannor. 2020. "Adaptive Trust Region Policy Optimization: Global Convergence and Faster Rates for Regularized MDPs." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Shao, Rulin, Shuyue Stella Li, Rui Xin, et al. 2025. "Spurious Rewards: Rethinking Training Signals in RLVR." *arXiv Preprint*.

Shao, Zhihong, Peiyi Wang, Qihao Zhu, et al. 2024. "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models." *arXiv Preprint arXiv:2402.03300*.

Shapley, Lloyd S. 1953. "Stochastic Games." *Proceedings of the National Academy of Sciences* 39 (10): 1095–100.

Shapley, Lloyd S. 1964. "Some Topics in Two-Person Games." In *Advances in Game Theory*. Annals of Mathematics Studies 52. Princeton University Press.

Sharma, Archit, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. 2020. "Dynamics-Aware Unsupervised Discovery of Skills." *International Conference on Learning Representations*.

Sharma, Mrinank, Meg Tong, Tomasz Korbak, et al. 2024. "Towards Understanding Sycophancy in Lan-

guage Models." *International Conference on Learning Representations*.

Shoham, Yoav, and Kevin Leyton-Brown. 2009. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

Silver, David, Hado van Hasselt, Matteo Hessel, et al. 2017. "The Predictron: End-to-End Learning and Planning." *Proceedings of the 34th International Conference on Machine Learning*.

Silver, David, Aja Huang, Chris J. Maddison, et al. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529 (7587): 484–89.

Silver, David, Thomas Hubert, Julian Schrittwieser, et al. 2018. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play." *Science* 362 (6419): 1140–44.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. "Deterministic Policy Gradient Algorithms." *Proceedings of the 31st International Conference on Machine Learning*, 387–95.

Silver, David, Julian Schrittwieser, Karen Simonyan, et al. 2017. "Mastering the Game of Go Without Human Knowledge." *Nature* 550 (7676): 354–59.

Silver, David, Satinder Singh, Doina Precup, and Richard S. Sutton. 2021. "Reward Is Enough." *Artificial Intelligence* 299: 103535.

Silver, David, and Joel Veness. 2010. "Monte-Carlo Planning in Large POMDPs." *Advances in Neural Information Processing Systems* 23.

Şimşek, Özgür, and Andrew G. Barto. 2008. "Skill Characterization Based on Betweenness." *Advances in Neural Information Processing Systems* 21.

Singh, Satinder P., Tommi Jaakkola, and Michael I. Jordan. 1994. "Learning Without State-Estimation in Partially Observable Markovian Decision Processes." *Proceedings of the Eleventh International Conference on Machine Learning*.

Singh, Satinder P., and Richard S. Sutton. 1996. "Reinforcement Learning with Replacing Eligibility Traces." *Machine Learning* 22: 123–58.

Singh, Satinder P., and Richard C. Yee. 1994. "An Upper Bound on the Loss from Approximate Optimal-Value Functions." *Machine Learning* 16 (3): 227–33.

Singh, Satinder, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. 2000. "Convergence Results for Single-Step on-Policy Reinforcement-Learning Algorithms." *Machine Learning* 38 (3): 287–308.

Singhal, Prasann, Tanya Goyal, Jiacheng Xu, and Greg Durrett. 2023. "A Long Way to Go: Investigating Length Correlations in RLHF." *arXiv Preprint*.

Skalse, Joar, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. "Defining and Characterizing Reward Hacking." *Advances in Neural Information Processing Systems* 35.

Slivkins, Aleksandrs. 2019. "Introduction to Multi-Armed Bandits." *Foundations and Trends in Machine Learning* 12 (1–2): 1–286.

Smallwood, Richard D., and Edward J. Sondik. 1973. "The Optimal Control of Partially Observable Markov Processes over a Finite Horizon." *Operations Research* 21 (5): 1071–88.

Smith, James E., and Robert L. Winkler. 2006. "The Optimizer's Curse: Skepticism and Postdecision Surprise in Decision Analysis." *Management Science* 52 (3): 311–22.

Smith, Trey, and Reid Simmons. 2004. "Heuristic Search Value Iteration for POMDPs." *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*.

Snell, Charlie, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. "Scaling LLM Test-Time Compute Optimally Can Be More Effective Than Scaling Model Parameters." *International Conference on Learning Representations*.

Sobel, Matthew J. 1982. "The Variance of Discounted Markov Decision Processes." *Journal of Applied Probability* 19 (4): 794–802.

Sokar, Ghada, Rishabh Agarwal, Pablo Samuel Castro, and Utku Evci. 2023. "The Dormant Neuron Phenomenon in Deep Reinforcement Learning." *Proceedings of the 40th International Conference on Machine Learning*.

Somani, Adhiraj, Nan Ye, David Hsu, and Wee Sun Lee. 2013. "DESPOT: Online POMDP Planning with Regularization." *Advances in Neural Information Processing Systems* 26.

Son, Kyunghwan, Daewoo Kim, Wan Ju Kang, David Earl Hostallero, and Yung Yi. 2019. "QTRAN: Learning to Factorize with Transformation for Cooperative Multi-Agent Reinforcement Learning." *Proceedings of the 36th International Conference on Machine Learning*.

Sondik, Edward J. 1978. "The Optimal Control of Partially Observable Markov Processes over the Infinite Horizon: Discounted Costs." *Operations Research* 26 (2): 282–304.

Spaan, Matthijs T. J., and Nikos Vlassis. 2005. "Perseus: Randomized Point-Based Value Iteration for POMDPs." *Journal of Artificial Intelligence Research* 24: 195–220.

Srikant, R., and Lei Ying. 2019. "Finite-Time Error Bounds for Linear Stochastic Approximation and TD Learning." *Proceedings of the 32nd Conference on Learning Theory*.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15: 1929–58.

Stachenfeld, Kimberly L., Matthew M. Botvinick, and Samuel J. Gershman. 2017. "The Hippocampus as a Predictive Map." *Nature Neuroscience* 20: 1643–53.

Stiennon, Nisan, Long Ouyang, Jeffrey Wu, et al. 2020. "Learning to Summarize with Human Feedback." *Advances in Neural Information Processing Systems* 33.

Stooke, Adam, Joshua Achiam, and Pieter Abbeel. 2020. "Responsive Safety in Reinforcement Learning by PID Lagrangian Methods." *Proceedings of the 37th International Conference on Machine Learning*.

Strehl, Alexander L., Lihong Li, and Michael L. Littman. 2009. "Reinforcement Learning in Finite MDPs: PAC Analysis." *Journal of Machine Learning Research* 10: 2413–44.

Strehl, Alexander L., Lihong Li, Eric Wiewiora, John Langford, and Michael L. Littman. 2006. "PAC Model-Free Reinforcement Learning." *Proceedings of the 23rd International Conference on Machine Learning*.

Strehl, Alexander L., and Michael L. Littman. 2008. "An Analysis of Model-Based Interval Estimation for Markov Decision Processes." *Journal of Computer and System Sciences* 74 (8): 1309–31.

Strens, Malcolm. 2000. "A Bayesian Framework for Reinforcement Learning." *Proceedings of the 17th International Conference on Machine Learning*.

Strubell, Emma, Ananya Ganesh, and Andrew McCallum. 2019. "Energy and Policy Considerations for Deep Learning in NLP." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Subramanian, Jayakumar, Amit Sinha, Raihan Seraj, and Aditya Mahajan. 2022. "Approximate Information State for Approximate Planning and Reinforcement Learning in Partially Observed Systems." *Journal of Machine Learning Research* 23.

Sunehag, Peter, Guy Lever, Audrunas Gruslys, et al. 2018. "Value-Decomposition Networks for Cooperative Multi-Agent Learning Based on Team Reward." *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems*.

Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton. 2013. "On the Importance of Initialization and Momentum in Deep Learning." *Proceedings of the 30th International Conference on Machine Learning*.

Sutton, Richard S. 1988. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning* 3 (1): 9–44.

Sutton, Richard S. 1990. "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming." *Proceedings of the Seventh International Conference on Machine Learning*, 216–24.

Sutton, Richard S. 1996. "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding." *Advances in Neural Information Processing Systems* 8.

Sutton, Richard S. 2019. *The Bitter Lesson*. Essay, http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press.

Sutton, Richard S., Marlos C. Machado, G. Zacharias Holland, et al. 2023. "Reward-Respecting Subtasks for Model-Based Reinforcement Learning." *Artificial Intelligence*.

Sutton, Richard S., Hamid Reza Maei, Doina Precup, et al. 2009. "Fast Gradient-Descent Methods for Temporal-Difference Learning with Linear Function Approximation." *Proceedings of the 26th International Conference on Machine Learning*.

Sutton, Richard S., A. Rupam Mahmood, and Martha White. 2016. "An Emphatic Approach to the Problem of Off-Policy Temporal-Difference Learning." *Journal of Machine Learning Research* 17.

Sutton, Richard S., David McAllester, Satinder Singh, and Yishay Mansour. 2000. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." *Advances in Neural Information Processing Systems* 12.

Sutton, Richard S., Doina Precup, and Satinder Singh. 1999. "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning." *Artificial Intelligence* 112 (1–2): 181–211.

Sutton, Richard S., Csaba Szepesvári, and Hamid Reza Maei. 2009. "A Convergent $O(n)$ Algorithm for Off-Policy Temporal-Difference Learning with Linear Function Approximation." *Advances in Neural Information Processing Systems* 21.

Swamy, Gokul, Sanjiban Choudhury, J. Andrew Bagnell, and Zhiwei Steven Wu. 2021. "Of Moments and Matching: A Game-Theoretic Framework for Closing the Imitation Gap." *Proceedings of the 38th International Conference on Machine Learning*.

Szepesvári, Csaba. 2010. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

Tamar, Aviv, Yonatan Glassner, and Shie Mannor. 2015. "Optimizing the CVaR via Sampling." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Tammelin, Oskari. 2014. "Solving Large Imperfect Information Games Using CFR+." *arXiv Preprint arXiv:1407.5042*.

Tan, Ming. 1993. "Multi-Agent Reinforcement Learning: Independent Vs. Cooperative Agents." *Proceedings of the Tenth International Conference on Machine Learning*.

Tang, Haoran, Rein Houthooft, Davis Foote, et al. 2017. "#Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning." *Advances in Neural Information Processing Systems* 30.

Tang, Yunhao, and Rémi Munos. 2025. "On a Few Pitfalls in KL Divergence Gradient Estimation for RL." *arXiv Preprint*.

Tassa, Yuval et al. 2018. "DeepMind Control Suite." *arXiv Preprint arXiv:1801.00690*.

Tesauro, Gerald. 1995. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM* 38 (3): 58–68.

Tesauro, Gerald, and Gregory R. Galperin. 1996. "On-Line Policy Improvement Using Monte-Carlo Search." *Advances in Neural Information Processing Systems* 9.

Tessler, Chen, Daniel J. Mankowitz, and Shie Mannor. 2019. "Reward Constrained Policy Optimization." *International Conference on Learning Representations*.

Thomas, Philip S. 2014. "Bias in Natural Actor-Critic Algorithms." *Proceedings of the 31st International Conference on Machine Learning*.

Thomas, Philip S., and Emma Brunskill. 2016. "Data-Efficient Off-Policy Policy Evaluation for Reinforcement Learning." *Proceedings of the 33rd International Conference on Machine Learning*.

Thomas, Philip S., Georgios Theocharous, and Mohammad Ghavamzadeh. 2015. "High-Confidence Off-Policy Evaluation." *Proceedings of the AAAI Conference on Artificial Intelligence*.

Thompson, William R. 1933. "On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples." *Biometrika* 25 (3–4): 285–94.

Thorndike, Edward L. 1911. *Animal Intelligence: Experimental Studies*. Macmillan.

Thrun, Sebastian, and Lorien Pratt, eds. 1998. *Learning to Learn*. Kluwer Academic Publishers.

Thrun, Sebastian, and Anton Schwartz. 1993. "Issues in Using Function Approximation for Reinforcement Learning." *Proceedings of the 1993 Connectionist Models Summer School*.

Tobin, Josh, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Todorov, Emanuel, Tom Erez, and Yuval Tassa. 2012. "MuJoCo: A Physics Engine for Model-Based Control." *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–33.

Tomar, Manan, Lior Shani, Yonathan Efroni, and Mohammad Ghavamzadeh. 2022. "Mirror Descent Policy Optimization." *International Conference on Learning Representations*.

Towers, Mark et al. 2024. "Gymnasium: A Standard Interface for Reinforcement Learning Environments." *arXiv Preprint arXiv:2407.17032*.

Tsitsiklis, John N. 1994. "Asynchronous Stochastic Approximation and Q-Learning." *Machine Learning* 16 (3): 185–202.

Tsitsiklis, John N. 2002. "On the Convergence of Optimistic Policy Iteration." *Journal of Machine Learning Research* 3: 59–72.

Tsitsiklis, John N., and Benjamin Van Roy. 1997. "An Analysis of Temporal-Difference Learning with Function Approximation." *IEEE Transactions on Automatic Control* 42 (5): 674–90.

Tucker, George, Surya Bhupatiraju, Shixiang Gu, Richard E. Turner, Zoubin Ghahramani, and Sergey

Levine. 2018. "The Mirage of Action-Dependent Baselines in Reinforcement Learning." *Proceedings of the 35th International Conference on Machine Learning*.

Uehara, Masatoshi, Jiawei Huang, and Nan Jiang. 2020. "Minimax Weight and Q-Function Learning for Off-Policy Evaluation." *Proceedings of the 37th International Conference on Machine Learning*.

Uesato, Jonathan, Nate Kushman, Ramana Kumar, et al. 2022. "Solving Math Word Problems with Process- and Outcome-Based Feedback." *arXiv Preprint arXiv:2211.14275*.

Uhlenbeck, George E., and Leonard S. Ornstein. 1930. "On the Theory of the Brownian Motion." *Physical Review* 36 (5): 823–41.

Vamplew, Peter, Benjamin J. Smith, Johan Källström, et al. 2022. "Scalar Reward Is Not Enough: A Response to Silver, Singh, Precup and Sutton (2021)." *Autonomous Agents and Multi-Agent Systems* 36 (2).

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems* 30.

Vezhnevets, Alexander Sasha, Simon Osindero, Tom Schaul, et al. 2017. "FeUdal Networks for Hierarchical Reinforcement Learning." *Proceedings of the 34th International Conference on Machine Learning*.

Vinyals, Oriol, Igor Babuschkin, Wojciech M. Czarnecki, et al. 2019. "Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning." *Nature* 575 (7782): 350–54.

Vlassis, Nikos, Michael L. Littman, and David Barber. 2012. "On the Computational Complexity of Stochastic Controller Optimization in POMDPs." *ACM Transactions on Computation Theory* 4 (4).

Voloshin, Cameron, Hoang M. Le, Nan Jiang, and Yisong Yue. 2021. "Empirical Study of Off-Policy Policy Evaluation for Reinforcement Learning." *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

Wainwright, Martin J. 2019. *Variance-Reduced Q-Learning Is Minimax Optimal*. arXiv preprint arXiv:1906.04697.

Wan, Yi, Abhishek Naik, and Richard S. Sutton. 2021. "Learning and Planning in Average-Reward Markov Decision Processes." *Proceedings of the 38th International Conference on Machine Learning*.

Wang, Che, Shuhan Yuan, Kai Shao, and Keith Ross. 2022. "On the Convergence of the Monte Carlo Exploring Starts Algorithm for Reinforcement Learning." *International Conference on Learning Representations*.

Wang, Guanzhi, Yuqi Xie, Yunfan Jiang, et al. 2023. "Voyager: An Open-Ended Embodied Agent with Large Language Models." *arXiv Preprint arXiv:2305.16291*.

Wang, Jane X., Zeb Kurth-Nelson, Dhruva Tirumala, et al. 2016. "Learning to Reinforcement Learn." *arXiv Preprint arXiv:1611.05763*.

Wang, Lingxiao, Qi Cai, Zhuoran Yang, and Zhaoran Wang. 2020. "Neural Policy Gradient Methods: Global Optimality and Rates of Convergence." *International Conference on Learning Representations*.

Wang, Peiyi, Lei Li, Zhihong Shao, et al. 2024. "Math-Shepherd: Verify and Reinforce LLMs Step-by-Step Without Human Annotations." *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*.

Wang, Ruosong, Dean P. Foster, and Sham M. Kakade. 2021a. "What Are the Statistical Limits of Offline RL with Linear Function Approximation?" *International Conference on Learning Representations*.

Wang, Ruosong, Dean P. Foster, and Sham M. Kakade. 2021b. "What Are the Statistical Limits of Offline RL with Linear Function Approximation?" *International Conference on Learning Representations*.

Wang, Xuezhi, Jason Wei, Dale Schuurmans, et al. 2023. "Self-Consistency Improves Chain of Thought Reasoning in Language Models." *International Conference on Learning Representations*.

Wang, Ziyu, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. "Dueling Network Architectures for Deep Reinforcement Learning." *Proceedings of the 33rd International Conference on Machine Learning*.

Watkins, Christopher J. C. H. 1989. "Learning from Delayed Rewards." PhD thesis, University of Cambridge.

Watkins, Christopher J. C. H., and Peter Dayan. 1992. "Q-Learning." *Machine Learning* 8 (3–4): 279–92.

Weaver, Lex, and Nigel Tao. 2001. "The Optimal Reward Baseline for Gradient-Based Reinforcement Learning." *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*.

Wei, Chen-Yu, Chung-Wei Lee, Mengxiao Zhang, and Haipeng Luo. 2021. "Last-Iterate Convergence of Decentralized Optimistic Gradient Descent/Ascent in Infinite-Horizon Competitive Markov Games." *Proceedings of the Thirty Fourth Conference on Learning Theory*.

Wei, Jason, Xuezhi Wang, Dale Schuurmans, et al. 2022. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *Advances in Neural Information Processing Systems* 35.

Weisz, Gellért, Philip Amortila, and Csaba Szepesvári. 2021. "Exponential Lower Bounds for Planning in MDPs with Linearly-Realizable Optimal Action-Value Functions." *Proceedings of the 32nd International Conference on Algorithmic Learning Theory*.

Welch, B. L. 1947. "The Generalization of 'Student's' Problem When Several Different Population Variances Are Involved." *Biometrika* 34 (1–2): 28–35.

Wen, Zheng, Doina Precup, Morteza Ibrahimi, Andre Barreto, Benjamin Van Roy, and Satinder Singh. 2020. "On Efficiency in Hierarchical Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Weng, Jiayi et al. 2022. "EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine." *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*.

Wiewiora, Eric. 2003. "Potential-Based Shaping and Q-Value Initialization Are Equivalent." *Journal of Artificial Intelligence Research* 19: 205–8.

Williams, Grady, Nolan Wagener, Brian Goldfain, et al. 2017. "Information Theoretic MPC for Model-Based Reinforcement Learning." *IEEE International Conference on Robotics and Automation*.

Williams, Ronald J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8 (3–4): 229–56.

Williams, Ronald J., and Leemon C. Baird. 1993. *Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions*. Northeastern University, College of Computer Science.

Williams, Ronald J., and Jing Peng. 1991. "Function Optimization Using Connectionist Reinforcement Learning Algorithms." *Connection Science* 3 (3): 241–68.

Wu, Cathy, Aravind Rajeswaran, Yan Duan, et al. 2018. "Variance Reduction for Policy Gradient with Action-Dependent Factorized Baselines." *International Conference on Learning Representations*.

Wu, Yifan, George Tucker, and Ofir Nachum. 2019a. "Behavior Regularized Offline Reinforcement Learning." *arXiv Preprint arXiv:1911.11361*.

Wu, Yifan, George Tucker, and Ofir Nachum. 2019b. "The Laplacian in RL: Learning Representations with Efficient Approximations." *International Conference on Learning Representations*.

Wu, Yue, Weitong Zhang, Pan Xu, and Quanquan Gu. 2020. "A Finite-Time Analysis of Two Time-Scale Actor-Critic Methods." *Advances in Neural Information Processing Systems* 33.

Wu, Yuhuai, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. 2017. "Scalable Trust-Region Method for Deep Reinforcement Learning Using Kronecker-Factored Approximation." *Advances in Neural Information Processing Systems* 30.

Wurman, Peter R., Samuel Barrett, Kenta Kawamoto, et al. 2022. "Outracing Champion Gran Turismo Drivers with Deep Reinforcement Learning." *Nature* 602 (7896): 223–28.

Xiao, Lin. 2022. "On the Convergence Rates of Policy Gradient Methods." *Journal of Machine Learning Research* 23.

Xie, Tengyang, Ching-An Cheng, Nan Jiang, Paul Mineiro, and Alekh Agarwal. 2021. "Bellman-Consistent Pessimism for Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 34.

Xu, Shusheng, Wei Fu, Jiaxuan Gao, et al. 2024. "Is DPO Superior to PPO for LLM Alignment? A Comprehensive Study." *Proceedings of the 41st International Conference on Machine Learning*.

Xu, Tengyu, Zhe Wang, and Yingbin Liang. 2020. "Improving Sample Complexity Bounds for (Natural) Actor-Critic Algorithms." *Advances in Neural Information Processing Systems* 33.

Yao, Shunyu, Dian Yu, Jeffrey Zhao, et al. 2023. "Tree of Thoughts: Deliberate Problem Solving with

Large Language Models." *Advances in Neural Information Processing Systems* 36.

Yao, Shunyu, Jeffrey Zhao, Dian Yu, et al. 2023. "ReAct: Synergizing Reasoning and Acting in Language Models." *International Conference on Learning Representations*.

Ye, Weirui, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. 2021. "Mastering Atari Games with Limited Data." *Advances in Neural Information Processing Systems* 34.

Ye, Yinyu. 2011. "The Simplex and Policy-Iteration Methods Are Strongly Polynomial for the Markov Decision Problem with a Fixed Discount Rate." *Mathematics of Operations Research* 36 (4): 593–603.

Yu, Chao, Akash Velu, Eugene Vinitsky, et al. 2022. "The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games." *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)* 35.

Yu, Huizhen. 2015. "On Convergence of Emphatic Temporal-Difference Learning." *Proceedings of the 28th Conference on Learning Theory*.

Yu, Qiying, Zheng Zhang, Ruofei Zhu, et al. 2025. "DAPO: An Open-Source LLM Reinforcement Learning System at Scale." *arXiv Preprint arXiv:2503.14476*.

Yu, Tianhe, Garrett Thomas, Lantao Yu, et al. 2020. "MOPO: Model-Based Offline Policy Optimization." *Advances in Neural Information Processing Systems* 33.

Yuan, Rui, Robert M. Gower, and Alessandro Lazaric. 2022. "A General Sample Complexity Analysis of Vanilla Policy Gradient." *Proceedings of the 25th International Conference on Artificial Intelligence and Statistics*.

Yue, Yang, Zhiqi Chen, Rui Lu, et al. 2025. "Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model?" *arXiv Preprint arXiv:2504.13837*.

Zanette, Andrea, Alessandro Lazaric, Mykel Kochenderfer, and Emma Brunskill. 2020. "Learning Near Optimal Policies with Low Inherent Bellman Error." *Proceedings of the 37th International Conference on Machine Learning*.

Zelikman, Eric, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. 2022. "STaR: Bootstrapping Reasoning with Reasoning." *Advances in Neural Information Processing Systems* 35.

Zhang, Chiyuan, Oriol Vinyals, Remi Munos, and Samy Bengio. 2018. "A Study on Overfitting in Deep Reinforcement Learning." *arXiv Preprint arXiv:1804.06893*.

Zhang, Kaiqing, Zhuoran Yang, and Tamer Başar. 2021. "Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms." In *Handbook of Reinforcement Learning and Control*. Springer.

Zhang, Zihan, Xiangyang Ji, and Simon S. Du. 2021. "Is Reinforcement Learning More Difficult Than Bandits? A Near-Optimal Algorithm Escaping the Curse of Horizon." *Proceedings of the 34th Conference on Learning Theory*.

Zhao, Tony Z., Vikash Kumar, Sergey Levine, and Chelsea Finn. 2023. "Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware." *Robotics: Science and Systems*.

Ziebart, Brian D. 2010. "Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy." PhD thesis, Carnegie Mellon University.

Ziebart, Brian D., Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. 2008. "Maximum Entropy Inverse Reinforcement Learning." *Proceedings of the AAAI Conference on Artificial Intelligence*, 1433–38.

Ziegler, Daniel M., Nisan Stiennon, Jeffrey Wu, et al. 2019. "Fine-Tuning Language Models from Human Preferences." *arXiv Preprint arXiv:1909.08593*.

Zinkevich, Martin, Michael Johanson, Michael Bowling, and Carmelo Piccione. 2007. "Regret Minimization in Games with Incomplete Information." *Advances in Neural Information Processing Systems* 20.

Zintgraf, Luisa, Kyriacos Shiarlis, Maximilian Igl, et al. 2020. "VariBAD: A Very Good Method for Bayes-Adaptive Deep RL via Meta-Learning." *International Conference on Learning Representations*.